Android nativo com Kotlin e MVVM

Simplificando técnicas avançadas

Casa do Código | alura

PAULO SALVATORE

Sumário

- ISBN
- Sobre o livro
- Agradecimentos
- 1. Iniciando a nossa jornada
- 2. Overview do projeto
- 3. Criando o projeto
- 4. Fluxo de navegação
- 5. Lista de criaturas
- 6. Data Binding
- 7. Adquirindo criaturas
- 8. Silhueta das criaturas
- 9. Visualizando criaturas
- 10. ReactiveX (RX)
- 11. Integrando com back-end
- 12. Encerramento

ISBN

Impresso: 978-85-5519-340-8

Digital: 978-85-5519-341-5

Arte da capa e ilustrações por Felipe Blanco (https://felipeblancoillustration.com/)

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Salvatore, Paulo
Android nativo com Kotlin e MVVM : simplificando técnicas avançadas / Paulo Salvatore. -- 1. ed. -- São Paulo : Aovs Sistemas de Informática, 2023.

Bibliografia.
ISBN 978-85-5519-340-8

1. Android (Programa de computador)
2. Aplicativos - Programas de computador
3. Aplicativos móveis I. Título.

23-158278 CDD-005.26

Índices para catálogo sistemático:

1. Android : Plataforma de desenvolvimento para aplicativos móveis : Programa de computador 005.26

Aline Graziele Benitez - Bibliotecária - CRB-1/3129

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Sobre o livro

Este livro é destinado a devs que desejam aprender a desenvolver aplicativos para Android nativo utilizando o Kotlin e diferentes bibliotecas. Ele apresenta um projeto de exemplo, o DevMon, que é inspirado na Pokédex (dispositivo eletrônico do universo de Pokémon™) e exibe criaturas mágicas com nomes inspirados em diversas tecnologias. O objetivo deste livro é ensinar como programar uma solução completa, além de abordar conceitos avançados do desenvolvimento nativo no Android, incluindo arquitetura, padrões e kits de ferramentas.

O livro cobre desde as bases do desenvolvimento de aplicativos para Android, passando por conceitos avançados e mostrando como aplicá-los de forma prática, com base em um projeto real. Além disso, o livro apresenta uma abordagem lúdica e criativa, utilizando criaturas mágicas como elementos de ensino para tornar a leitura mais agradável e divertida.

Este livro é uma excelente opção para quem quer entender um pouco mais do ecossistema de desenvolvimento Android, com uma abordagem prática, baseada em projetos reais, de forma lúdica e criativa. O projeto DevMon permite aplicar os conceitos aprendidos de forma concreta e visualizar o resultado final do desenvolvimento. Isso ajuda a solidificar o aprendizado e torna o conteúdo mais facilmente aplicável na prática.

Para quem é este livro

Este livro é destinado a pessoas que querem ter um aspecto geral do desenvolvimento avançado com Android nativo, usando Kotlin

como linguagem de programação.

Não é necessário um conhecimento profundo sobre Android para produzir os projetos; entretanto, é importante que já tenha familiaridade com lógica de programação e experiência com alguma linguagem de programação.

Pré-requisitos

A principal motivação para a criação deste livro é desmistificar o desenvolvimento de aplicativos Android, utilizando a plataforma nativa em Java/Kotlin. É importante ressaltar que, quando optamos por desenvolver nesse ambiente, algumas considerações devem ser feitas para entender o momento em que a tecnologia se encontra e os motivos pelos quais utilizaremos as abordagens presentes neste livro.

Como este livro tem o objetivo de trazer diversos aspectos do desenvolvimento avançado para a plataforma Android, assumiremos que alguns conceitos básicos da plataforma Android já foram vistos, mesmo que de forma superficial. Os tópicos com os quais você já deve ter alguma familiaridade antes de ler este livro são:

Conhecimentos específicos de Android:

- Activities (classes e layout XML);
- Fragments;
- Intents;
- Estrutura de arquivos e pastas do Android;
- Criação e manipulação de elementos de UI (Views como TextView, EditText, Button, Layouts etc.).

Conhecimentos gerais de programação:

- Lógica de programação;

- Sintaxe básica com alguma linguagem de programação;
- Conhecimentos básicos em programação orientada a objetos.

Caso esses conhecimentos sejam novos para você, recomendamos que assista a um conteúdo introdutório sobre Android ou programação antes de ler este livro.

No canal do YouTube do Paulo Salvatore, há uma playlist de Android com vídeos introdutórios que devem auxiliar você nesse processo.

Playlist *Android Nativo com Kotlin*:
https://www.youtube.com/playlist?list=PLzFYWfRVHp5LxSqfYXvDAq-76lyAFqrMM

Figura -1.1: Playlist Android - YouTube.

Considerações

Quando estamos construindo uma aplicação para Android nativo, precisamos considerar três categorias principais: **Linguagem de programação**, **Arquitetura** e **Kit de ferramentas**.

Ao considerarmos esses três tópicos, conseguimos entender os motivos fundamentais de o desenvolvimento Android ser do jeito que ele é hoje, além de compreendermos como usar isso a nosso favor, considerando as linguagens de programação existentes, os modelos de arquitetura disponíveis e o kit de ferramentas, que cada vez mais evolui.

No primeiro e no segundo capítulo deste livro, detalhamos esses três tópicos e a importância de cada um para o universo Android.

A partir do terceiro capítulo, criamos o projeto e começamos a construir as suas fundações: fluxo de navegação entre telas; construção da lista de criaturas; e preparação do *data binding*.

Com as fundações preparadas, é hora de trazer mais comportamento para a aplicação e aplicar os conceitos de arquitetura na prática. Entre o sétimo e o nono capítulo, implementamos a aquisição de criaturas, a função de esconder as criaturas ainda desconhecidas e a de visualizar mais informações de cada criatura individualmente.

Por fim, nos dois últimos capítulos, introduzimos a biblioteca RX, ensinando a trabalhar com fluxos de dados síncronos e assíncronos, e concluímos com a integração a uma API web, obtendo dados das criaturas a partir de um servidor online.

Sobre o autor

Professor e desenvolvedor, comecei a ~~programar~~ copiar e colar código por volta dos 12 anos, criando meus próprios servidores de Tibia utilizando Lua e C++.

Iniciei a faculdade aos 22. Tive o meu primeiro emprego como dev em 2016, aos 23, e me tornei CTO aos 24. Atuando como professor desde março de 2017, estou sempre em busca de compartilhar o conhecimento que adquiri ao longo dessa jornada. Atualmente, leciono cursos gratuitos no Samsung Ocean e faço parte da FIAP, na graduação de Jogos Digitais e no MBA de Gestão da TI (https://oceanbrasil.com/ e https://fiap.com.br/).

Formado em Jogos Digitais pela FIAP, onde também fiz um MBA em Gestão Estratégica de Negócios, para sair um pouco da zona de conforto tech. Colaborei em um artigo científico publicado na revista *Water* sobre o uso da realidade virtual para engajar comunidades a entenderem consequências da mudança climática, em específico o aumento do nível dos oceanos (https://mdpi.com/2073-4441/13/9/1142).

Em 2018, cofundei a Virtual Planet, startup localizada na Califórnia que utiliza tecnologias imersivas (como a realidade virtual) para comunicar melhor os problemas relacionados à mudança climática, como o aumento do nível dos oceanos e da quantidade de incêndios florestais, além de jogos que tratam sobre o tema saúde mental, animações 3D que falam sobre áreas marinhas protegidas, entre diversos outros projetos (https://virtualplanet.tech/).

Escritor, muito feliz em publicar pela Casa do Código o meu primeiro livro, que tem o objetivo de apresentar de forma dinâmica e lúdica a criação de apps Android com Kotlin, introduzindo o universo do DevMon e guiando as implementações em cima de um projeto

simples e que simula a realidade (https://fabricadesinapse.github.io/DevMon/).

Atualmente, estou em busca de ampliar os meus projetos de educação e adquirir novas experiências no mundo da programação.

Agradecimentos

Este projeto começou uns bons anos antes da sua publicação, quando comecei a lecionar sobre a tecnologia Android e decidi compartilhar meus conhecimentos através de um livro. Depois de concluir a revisão, decidi reestruturá-lo e introduzir o universo mágico de DevMon, que estará presente em diversos conteúdos educacionais produzidos por mim.

Gostaria de agradecer aos que me ajudaram na criação deste livro de desenvolvimento Android. Em especial, quero agradecer à **Vivian Matsui** e à **Casa do Código**, que foram fundamentais para o sucesso do projeto, me ajudando a aprimorar a escrita e a transmitir conhecimentos de maneira eficiente. Obrigado pela paciência em todo o processo.

Quero agradecer ao meu professor **Heider Lopes**, que me ensinou Android e me inspirou em como ser um professor de tecnologia. Aos meus professores na FIAP e na vida, **Bruno Carvalho**, **Marcel Casarini**, **Renê Oliveira** e **Thiago Coser**, que me inspiraram a compartilhar os conhecimentos adquiridos ao longo dessa jornada.

Também agradeço ao **Samuel Araújo** e ao **Pablo Rodrigues**, pela ajuda na revisão dos conteúdos técnicos. Aos meus colegas de faculdade, alunos, amigos e familiares, agradeço por todo o apoio e incentivo durante a realização deste sonho em forma de livro.

Obrigado, também, aos leitores e leitoras que confiaram em mim e adquiriram este livro.

Capítulo 1
Iniciando a nossa jornada

Figura 1.1: Logo DevMon.

Durante os capítulos deste livro, desenvolveremos o aplicativo do DevMon, um universo mágico habitado por criaturas mágicas com nomes inspirados em linguagens de programação, frameworks e diversas outras tecnologias. A aplicação é inspirada na Pokédex, que exibe as criaturas que já foram reveladas ou apenas a silhueta para criaturas desconhecidas. Para quem nunca ouviu falar, a Pokédex é um dispositivo eletrônico que serve como catálogo das criaturas mágicas que existem no universo de Pokémon™.

Pegue um café, um suco, um chá ou qualquer que seja a sua bebida favorita, e prepare-se para ler este conteúdo, que foi preparado com o maior amor!

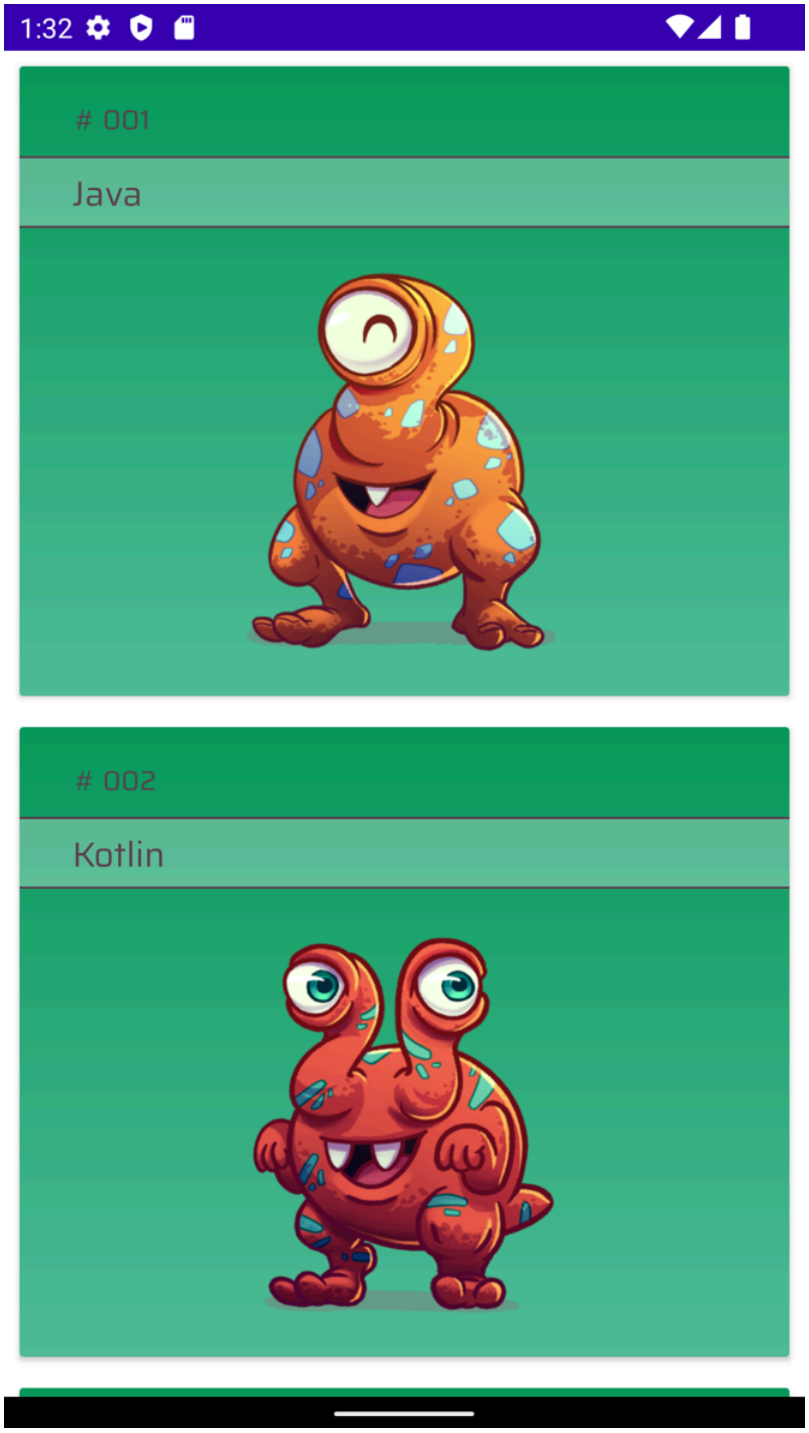


Figura 1.2: Exemplo de uma tela do aplicativo.

Conhecendo o projeto

DevMon é um projeto com o objetivo de ensinar devs a programar uma solução completa em diversos frameworks. Nesta versão, apresentaremos conceitos avançados do desenvolvimento nativo em Android, utilizando Kotlin como linguagem de programação.

O universo do DevMon é habitado por criaturas mágicas que surgem a partir de alguns ovos místicos, encontrados em diversos locais da natureza. Para que as criaturas evoluam, é necessário cuidar delas, brincar, alimentá-las e treiná-las.

No começo de cada capítulo, a silhueta de uma ou mais criaturas é apresentada. Durante o capítulo, ou no final dele, as criaturas são reveladas, até finalmente conhecermos todas elas.

A primeira criatura que será revelada é nomeada em homenagem a uma linguagem de programação amplamente utilizada, que existe há vários anos e que evoluiu muito ao longo do tempo, além de poder ser executada em praticamente qualquer tipo de dispositivo, por mais simples que ele seja.

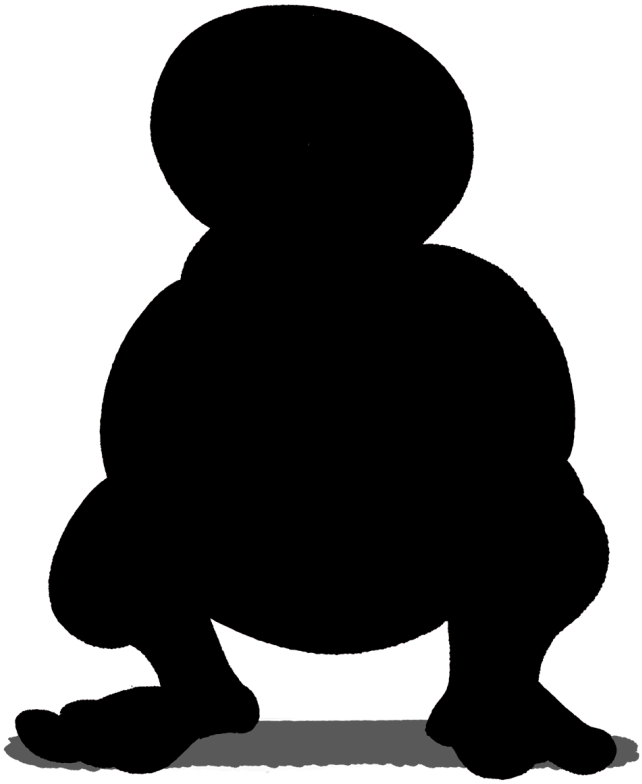


Figura 1.3: Qual é o DevMon #1?

1.1 Linguagem de programação

O framework para desenvolvimento de aplicações que rodam de forma nativa no Android é construído com a linguagem Java, enquanto o sistema operacional em si foi desenvolvido com a linguagem C. Portanto, sempre que um aplicativo estiver rodando no smartphone, o arquivo binário da aplicação, geralmente na extensão `.apk` ou `.aab`, será executado pela JVM (*Java Virtual Machine*), que também foi escrita com a linguagem C.

É importante ter um conhecimento da linguagem Java para desenvolver as aplicações que rodam no Android, pois esse é o código que será executado de fato. No entanto, o Java é uma linguagem que foi lançada no ano de 1996 e recebeu muitas

atualizações e melhorias desde então. Quando uma linguagem recebe uma atualização, novas funcionalidades são incluídas com diversos objetivos, tais como simplificar a forma de declaração, adicionar funcionalidades novas que não eram possíveis anteriormente, otimizar a performance, entre diversos outros.

No Android, para que o suporte a uma versão nova do Java ocorra, é necessário aguardar que esse desenvolvimento seja feito por parte da equipe que mantém o framework Android funcionando. Por isso, precisamos assumir que várias funcionalidades novas inseridas na linguagem Java **não estarão disponíveis para utilização**.

Com isso em mente, podemos assumir que escrever um código de uma aplicação Android em Java fará com que tenhamos que utilizar declarações que são mais *verbosas*, ou seja, é necessário escrever mais código para desenvolver uma funcionalidade relativamente simples.

Apesar de o Java ter evoluído muito como linguagem ao longo dos anos, o Android ainda se encontra preso em versões antigas e declarações que não evoluem tão rápido, principalmente porque temos dispositivos Android espalhados ao redor do mundo que ainda rodam versões antigas do sistema operacional.

Por esses motivos, surgem outras linguagens que tentam resolver esse problema mantendo o suporte às versões antigas, mas permitindo a inserção de funcionalidades mais poderosas e que reduzem o trabalho necessário e a quantidade de código para construir uma aplicação: são as **linguagens transpiladas**.

Linguagens transpiladas

As linguagens transpiladas, por definição, traduzem o código que foi escrito em uma linguagem para uma outra linguagem, da mesma forma que as linguagens de programação compiladas, como o Java, traduzem um código textual para um código binário, que será executado posteriormente.

As principais linguagens que transpilam para o código que roda na JVM hoje são:

- Groovy
- Kotlin
- Scala
- Clojure

Portanto, qualquer linguagem de programação que seja capaz de produzir um arquivo que roda na JVM é compatível com o desenvolvimento Android.

Sabendo disso, e considerando todos os problemas mencionados anteriormente, a comunidade de desenvolvimento Android começou uma busca por alternativas à utilização de Java como a principal linguagem, construindo aplicações com linguagens alternativas muito antes de qualquer suporte oficial ser adicionado.

Kotlin

Percebendo o caminho que a comunidade estava trilhando, o Google, em 2017, anunciou suporte oficial à linguagem de programação Kotlin, uma linguagem relativamente nova, cujo projeto foi iniciado em 2010, divulgado pela primeira vez em 2011 e só se tornou open source em 2012.

Um dos principais motivos pela decisão é a empresa por trás da linguagem, a JetBrains, que desenvolveu e mantém o Android Studio (a IDE na qual aplicações Android são desenvolvidas), além de ter desenvolvido outras IDEs famosas que usam a plataforma IntelliJ, como IDEA, PyCharm, PhpStorm, WebStorm, Rider, entre diversas outras.

Outro fator bem importante é a interoperabilidade com o Java. Sim! O Kotlin conversa normalmente com o Java! Isso significa que é possível iniciar a escrever Kotlin em um projeto que esteja feito 100% em Java e conseguir executar suas bibliotecas normalmente. Além disso, temos fatores cruciais como:

- **Compatibilidade:** totalmente compatível com `JDK 6`, o que garante a execução em *devices* mais antigos;
- **Desempenho:** um aplicativo Kotlin é tão rápido quanto um aplicativo em Java, às vezes até ganhando em termos de velocidade;
- **Tempo de compilação:** o mesmo caso que o desempenho, velocidade igual ou superior (em alguns casos) ao Java;
- **Curva de aprendizado:** para quem desenvolve em Java, é extremamente fácil iniciar em Kotlin. Garanto que, no primeiro projeto usando a linguagem, você já se sentirá apaixonado por ela e saberá usar até os recursos avançados da linguagem — e com aquele sentimento: "Por que ninguém me mostrou isso antes?".

Nos próximos capítulos, apresentaremos algumas características da linguagem para auxiliar no entendimento inicial de como desenvolver usando Kotlin.

1.2 Arquitetura

Uma das coisas mais importantes (se não a mais importante) de um software é a arquitetura, assunto que é muito discutido entre a comunidade dev. Para entender os caminhos que o Android seguiu, precisamos conhecer um pouco da evolução dos padrões de arquitetura **MVC**, **MVP** e **MVVM**. Conhecer a evolução e a importância de cada padrão é fundamental, pois nos permite entender os motivos de determinados padrões funcionarem melhor em um projeto Android.

Dentre os três padrões mencionados, todos eles possuem algo em comum: as letras **V** e **M**, que representam ***View*** e ***Model***, respectivamente.

Sabemos que, no Android, toda aplicação possui telas, criadas a partir da classe `Activity`, contendo um arquivo de *layout* em `XML` e, possivelmente, possuindo `Fragments` que promovem a reutilização de telas e que separam melhor pequenas funcionalidades de um sistema. Como essas telas exibem informações para o usuário, costumamos chamá-las de **Views**, pois permitem que a informação seja visualizada.

Para que uma aplicação cumpra seus objetivos, muitas vezes é necessário utilizar recursos mais completos (e mais complexos), como integrar com APIs na internet, usar banco de dados, exibir notificações, reproduzir sons, exibir imagens etc. Para que isso ocorra, precisamos criar novos arquivos que contenham o código que fará a funcionalidade desejada, criando uma necessidade de organização desses códigos que também tenha como objetivo facilitar a manutenção deles.

Quando essas funcionalidades exigem trabalhar e manusear dados como informações do usuário, lista de produtos disponíveis ou links das imagens, costumamos guardar os códigos que resolvem isso em uma pasta, a qual chamamos de ***Model***, pois contém toda a lógica necessária para resolver esse tipo de problema.

Como esse é um problema que todo software possui, independente da sua linguagem, existem soluções aparecendo há muito tempo. O MVC, por exemplo, foi formulado na década de 1970, quando ainda "era tudo mato". Já o MVP foi formulado no ano de 1996, quando a internet estava dando seus primeiros passos. Por fim, o MVVM teve sua primeira aparição em 2005.

Devido ao fato de que cada framework tem as suas próprias características, objetivos e problemas específicos a serem resolvidos, o padrão de arquitetura escolhido pode variar. Além disso, novos problemas que surgem ao longo do tempo podem fazer com que a comunidade migre de padrão de arquitetura e tente utilizar outros formatos.

No caso dos três padrões mencionados, temos uma camada extra que conecta o **Model** à **View**. A diferença é que, em cada padrão, essa camada extra possui características diferentes e todas possuem seus prós e contras. Para entender um pouco mais sobre essas diferenças, vamos ver como cada padrão funciona separadamente.

MVC

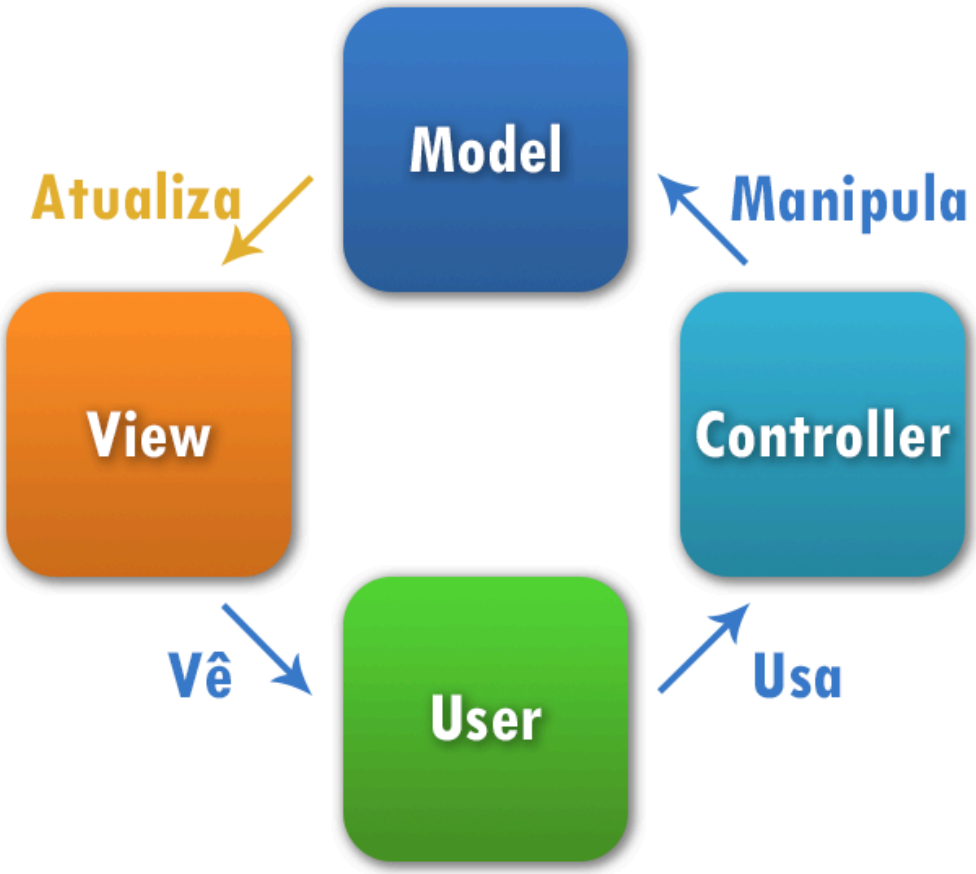


Figura 1.4: Diagrama MVC.

O MVC, *Model-View-Controller*, é um padrão de arquitetura que sintetiza a maneira como as informações de uma aplicação devem estar relacionadas, colocando o usuário em uma das pontas e ditando exatamente o que cada parte do software deve fazer para que o conteúdo chegue até ele — tudo isso utilizando boas práticas de programação, promovendo a reutilização de código e visando a um trabalho mais eficiente.

Essa abordagem surgiu em 1979, na Xerox PARC, e é uma das mais conhecidas. É utilizada em diversas plataformas de desenvolvimento e contempla diversas linguagens de programação, principalmente nas aplicações feitas para a web.

A ***View*** é o componente pelo qual o usuário visualiza todas as informações carregadas pelos componentes citados anteriormente. Uma característica é que ela não pode conter nenhuma lógica e não deve saber de nada do que está acontecendo nos bastidores da aplicação. Ela apenas recebe a informação e a exibe da melhor maneira possível. Ela é atualizada conforme os dados recebidos do *Model*, e geralmente essa atualização é feita pelo *Controller*.

O ***Controller***, por sua vez, é responsável por solicitar informações do *Model* conforme a necessidade do usuário. Ele também solicitará atualizações no *Model*, informando que um cadastro foi atualizado, por exemplo, ou que um registro precisa ser removido. Todos esses gatilhos são realizados a partir da interação do usuário com o sistema, onde o *Controller* recebe diretamente essa interação e a repassa para o local necessário.

Por fim, o ***Model*** é responsável pela comunicação com os dados, independente de onde eles estão. Ele vai receber notificações do *Controller* conforme a necessidade desses dados, vai buscá-los, vai realizar alguma lógica caso haja necessidade e retornar a informação pronta para ser utilizada.

Uma frase bem conhecida nesse padrão é "*Fat models and skinny controllers*", que significa "*models* inflados e *controllers* magros", deixando claro que a camada *Model* costuma ter mais conteúdo do que a camada *Controller* .

A ideia dessa frase é que a principal lógica dos dados deve estar no *Model* e não no *Controller*.

No Android:

- o *Model* seria composto de classes de acesso ao banco de dados ou de acesso a APIs na web;
- a *View* seria o layout `XML` que exibe os dados diretamente inseridos pela classe da `Activity`;
- e o *Controller* seria nossa classe `Activity` e qualquer classe de comunicação que solicitasse informações ao *Model*.

A abordagem pode variar em diversos casos, mas essa seria uma representação básica, na qual não temos uma separação de fato entre *Controller* e *View*, onde nossa `Activity` ou `Fragment` atuava muitas vezes em ambas as funções.

MVP

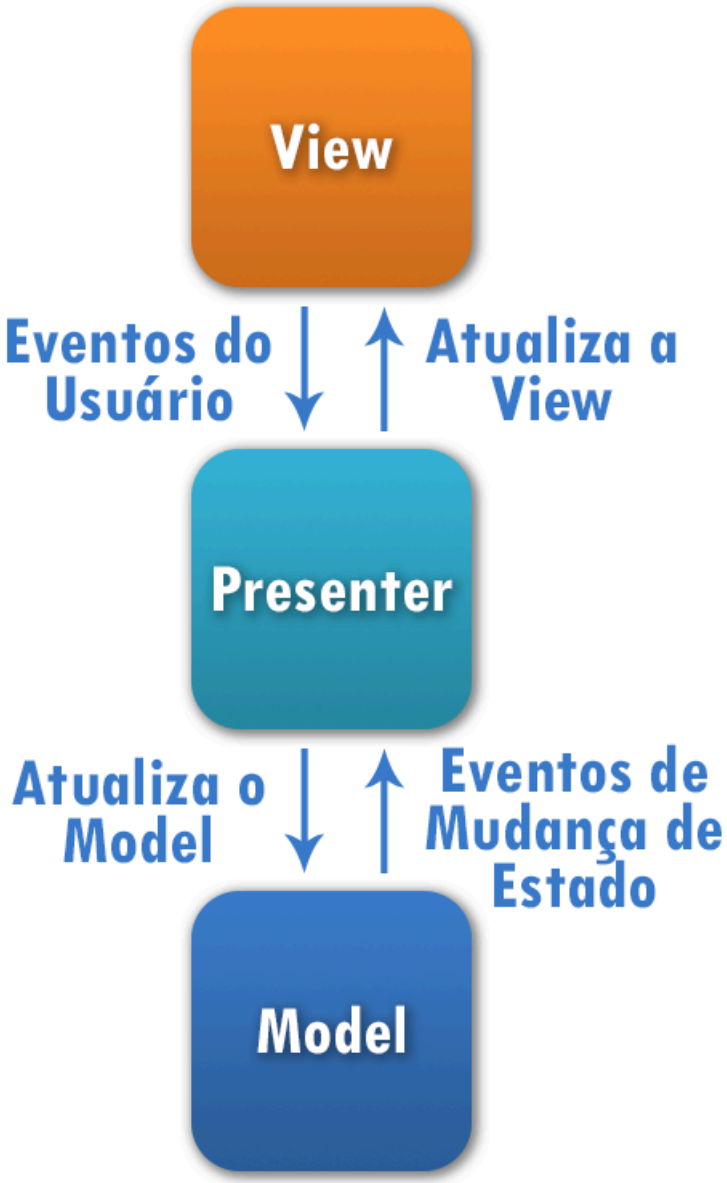


Figura 1.5: Diagrama MVP.

O MVP, *Model-View-Presenter*, é um padrão de arquitetura que deriva do MVC e que é usado na maioria das vezes na construção de interfaces de usuário, como é o caso do Android. Nessa abordagem, temos a substituição do *Controller* pelo *Presenter*, que assume a função de "homem-do-meio". Toda a lógica será feita pelo *Presenter*, que basicamente responde às ações da UI e controla a interação entre a *View* e o *Model*.

A diferença entre o *Controller* e o *Presenter* é que o *Presenter* fica mais desacoplado da *View*, o que aumenta o nível de abstração e separa melhor as responsabilidades.

Nessa arquitetura, o *Model* e a *View* são idênticos ao apresentado no MVC. O *Presenter*, que entra para substituir o *Controller*, é responsável por intermediar a relação entre a *View* e o *Model*, garantindo que qualquer comunicação necessária passe por ele. Ele recebe os dados fornecidos pelo *Model*, formata e envia para serem exibidos na *View*. Em algumas implementações, é comum criarmos um novo componente chamado ***Supervising Controller***, que vai cuidar de casos em que é necessário recuperar ou persistir o *Model*.

No Android:

- o *Model* mantém as mesmas classes do MVC;
- a *View* passa a possuir a classe da `Activity`, além dos layouts que já existiam no padrão MVC, e também passa a implementar uma interface *View* que será reconhecida pelo *Presenter*;
- o *Presenter* são novas classes que implementam uma interface *Presenter* que vão atualizar o *Model* para solicitar os dados, além de guardar a referência da interface *View* implementada. É preferível que o *Presenter* não possua nenhuma referência ou código relacionado à API do Android.

O MVP se tornou muito popular entre desenvolvedores Android com o passar do tempo; porém, iniciar o desenvolvimento nessa abordagem requer muita escrita de código repetitivo necessário (conhecido como *boilerplate*), e algumas práticas, como guardar a referência da *View* em questão dentro do *Presenter*, dificultam alguns processos dentro dessa arquitetura, principalmente quando temos um *Presenter* muito grande e difícil de quebrar em pedaços.

MVVM

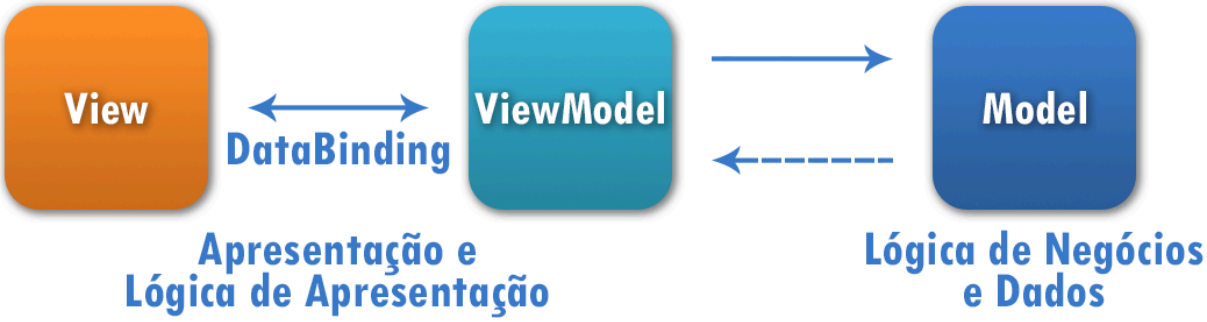


Figura 1.6: Diagrama MVVM.

O MVVM, *Model-View-ViewModel*, também conhecido como *Model-View-Binder*, é uma das abordagens mais atuais em termos de arquitetura de software, principalmente na comunidade Android, recebendo cada vez mais atenção de devs e pessoas adeptas da *Clean Architecture*.

Essa abordagem tem como característica facilitar a separação de conceitos no desenvolvimento de uma interface de usuário. Ela foi desenvolvida para utilizar funções conhecidas como *Data Bindings*, que facilitam uma melhor abstração da *View* do resto do projeto, pois remove boa parte do código de atualização dos elementos da UI. Com isso, elimina-se a necessidade de designers de UX escreverem esses códigos, permitindo que uma linguagem de marcação seja utilizada para isso.

A separação dessas tarefas foi um dos pontos pensados durante o desenvolvimento dessa arquitetura: o objetivo era acelerar o processo de desenvolvimento de uma aplicação tanto em projetos grandes quanto em projetos pequenos.

Nessa arquitetura, o *Model* é idêntico ao apresentado no MVC e no MVP. A *View* ganha uma nova funcionalidade de *bind* em variáveis "observáveis" e ações presentes no *View Model* de uma maneira mais flexível. O *View Model*, que entra para substituir o *Controller* e o *Presenter*, é responsável por envolver o *Model*, preparando variáveis "observáveis" que são necessárias para a *View*. Ele também funciona como uma ponte para a *View* enviar eventos ao

Model. Uma característica importante desse modelo é que o *View Model* não possui referência da *View*, permitindo com mais facilidade que um *View Model* sirva várias *views*.

No Android:

- o *Model* mantém a mesma base do MVC;
- a única diferença da *View* é que ela não possui mais uma interface *View* implementada e passa a ter declarações de observações às variáveis do *View Model*;
- e o *View Model* possuirá uma implementação da interface *View Model* que está ciente do ciclo de vida da `Activity` ou `Fragment` e ficará responsável pela declaração e atualização das variáveis "observáveis" que serão observadas pela *View*.

O MVVM é um padrão de arquitetura que vem ganhando cada vez mais espaço entre a comunidade Android, recomendado atualmente pelo Google e que ganhou muita força após a Google I/O '17, com o anúncio dos *Architecture Components* dentro do **Android Jetpack** (falaremos sobre ele mais à frente). Eles facilitaram muito a sua implementação e permitiram que aplicações de altíssimo nível de arquitetura fossem criadas do zero sem muito esforço inicial, algo que as abordagens anteriores não tinham como principal característica.

Qual arquitetura utilizar no Android?

No Android (em qualquer framework, na verdade), é possível utilizar qualquer um dos três padrões mencionados, como também utilizar (ou criar) um padrão completamente novo. Entretanto, conceber esses padrões que a comunidade não costuma utilizar pode resultar em muita escrita de código *boilerplate* — e em alguns problemas desconhecidos que podem surgir pelo caminho. Por conta disso, no Android, o MVC foi um padrão amplamente utilizado no início, mas rapidamente houve uma migração para o MVP, presente em várias aplicações até hoje.

Como mencionado anteriormente, o MVVM vem ganhando muita força por conta do Android Jetpack, que consiste em um **kit de ferramentas** extremamente poderoso que facilita a adoção ao padrão de arquitetura.

1.3 Kit de ferramentas

O kit de ferramentas é o último ponto que precisamos considerar no desenvolvimento Android, pois utilizar bibliotecas prontas nos permite focar apenas nos problemas específicos da aplicação que estamos desenvolvendo, o que facilita e acelera o processo de construção.

Ter acesso a ferramentas prontas faz com que as decisões sobre qual caminho seguir sejam mais fáceis, pois elas geralmente apontam caminhos que a comunidade está trilhando, para que todos fiquem no mesmo barco. Portanto, optar por utilizar uma ferramenta presente no Android Jetpack costuma ser uma ótima decisão.

Android Jetpack

O **Android Jetpack** é um conjunto de bibliotecas, ferramentas e guias de arquitetura que facilitam e aceleram a construção de bons aplicativos. Ele fornece uma boa base de códigos e funcionalidades, como podemos ver na imagem a seguir.

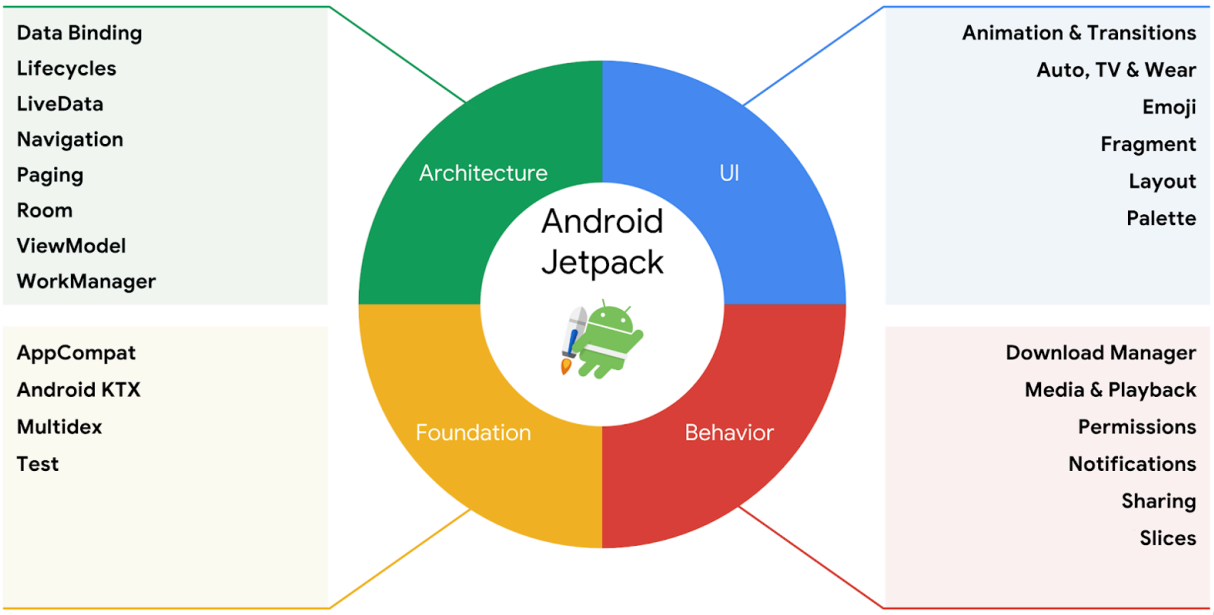


Figura 1.7: Componentes presentes no Jetpack anunciados na Google I/O '17 e '18.

Confira a lista completa de bibliotecas disponíveis em: https://developer.android.com/jetpack/androidx/explorer

Figura 1.8: Lista de bibliotecas do Jetpack - Documentação do Android.

Ao longo dos anos, novas bibliotecas são introduzidas, e o cenário evolui constantemente. Em 2017, tivemos a apresentação da nova

biblioteca conhecida como **Android Architecture Components**, com diversas features poderosas para o desenvolvimento Android em diversos aspectos.

Os principais componentes que beneficiam o uso do MVVM, por exemplo, são: **Data Binding**, **Lifecycle**, **ViewModel** e **LiveData**.

A biblioteca tem como objetivo auxiliar os desenvolvedores a construir aplicações que sejam robustas, testáveis e fáceis de manter. Resumidamente, ela nos ajuda a lidar melhor com a persistência de dados em eventos que envolvem ciclos de vida e alterações de configuração, além de nos ajudar a criar um aplicativo com arquitetura melhor e evitar classes cheias de repetição de códigos que são difíceis de manter e testar.

Próximos passos

Agora que esclarecemos um pouco as coisas, e com as devidas considerações feitas, podemos avançar para o próximo capítulo, onde daremos um *overview* sobre o projeto que servirá como base durante todo o livro, passando por diversos conceitos importantíssimos e apresentando a stack de tecnologia que utilizaremos.

Por último, mas não menos importante, o DevMon #1 se chama **Java**, em homenagem à linguagem de programação que está presente em muitos softwares, independente da plataforma em que eles estejam. A JVM permite a execução de código Java em sistemas embarcados, que vão desde smartphones e desktops a geladeiras e outros dispositivos.

URL da imagem: https://i.imgur.com/PHfd1h2.png.

Figura 1.9: O DevMon #1 se chama Java!

Capítulo 2
Overview do projeto

Certifique-se de que está com o Android Studio na última versão disponível e faça o clone do repositório deste livro, disponível no GitHub, através do link a seguir:
https://github.com/FabricaDeSinapse/livro-android-casa-do-codigo

Figura 2.1: Repositório no GitHub.

Utilizaremos esse repositório como um guia durante todo o livro. Ele contém pastas com cada uma das etapas descritas nos capítulos. Isso deve ajudar você quando estiver com problema na implementação de algum capítulo específico, sendo possível comparar como deverá ficar o projeto ao final de cada etapa. Além disso, você pode desenvolver cada capítulo individualmente, focando apenas naqueles que mais lhe interessam, sem a necessidade de construir todo o projeto do início ao fim.

Para iniciar este capítulo em grande estilo, a segunda criatura que será revelada é a evolução da criatura Java, que apresentamos no

capítulo anterior. Ela também é nomeada em homenagem a uma linguagem de programação muito poderosa e que vem ganhando cada vez mais destaque. Sua principal característica é ser 100% interoperável com o Java, além de possuir uma forma de transpilar código para JavaScript. Já sabe qual é o nome dela?

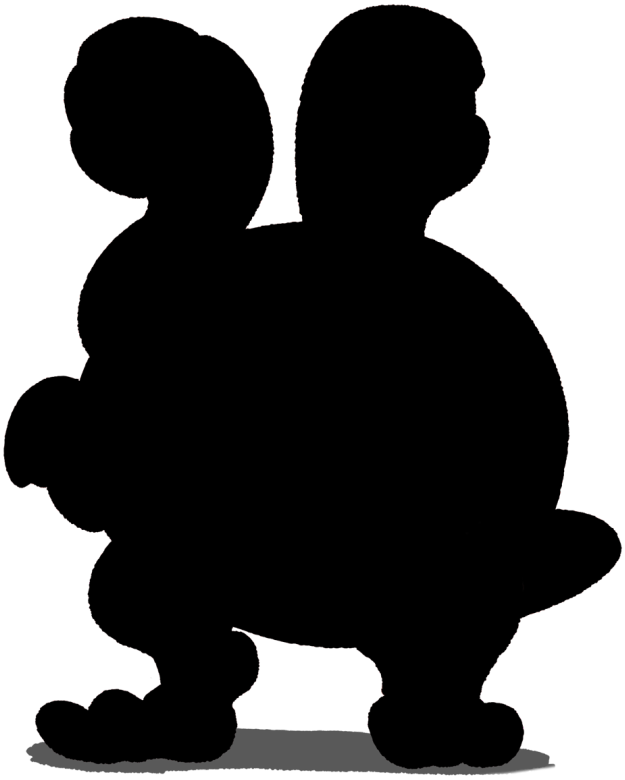


Figura 2.2: Qual é o DevMon #2?

Arquitetura — MVVM

Seguindo as tendências do universo Android, implementaremos o DevMon utilizando o padrão MVVM, aplicando diversos conceitos avançados como:

- Princípios **SOLID**;
- **IoC** - *Inversion of Control* (Inversão de controle);
- **Injeção de dependência**;
- ***Loose coupling*** (Baixo acoplamento);
- Alguns conceitos de ***Clean Architecture***;

- **Programação Funcional Reativa** (RX);
- entre outros.

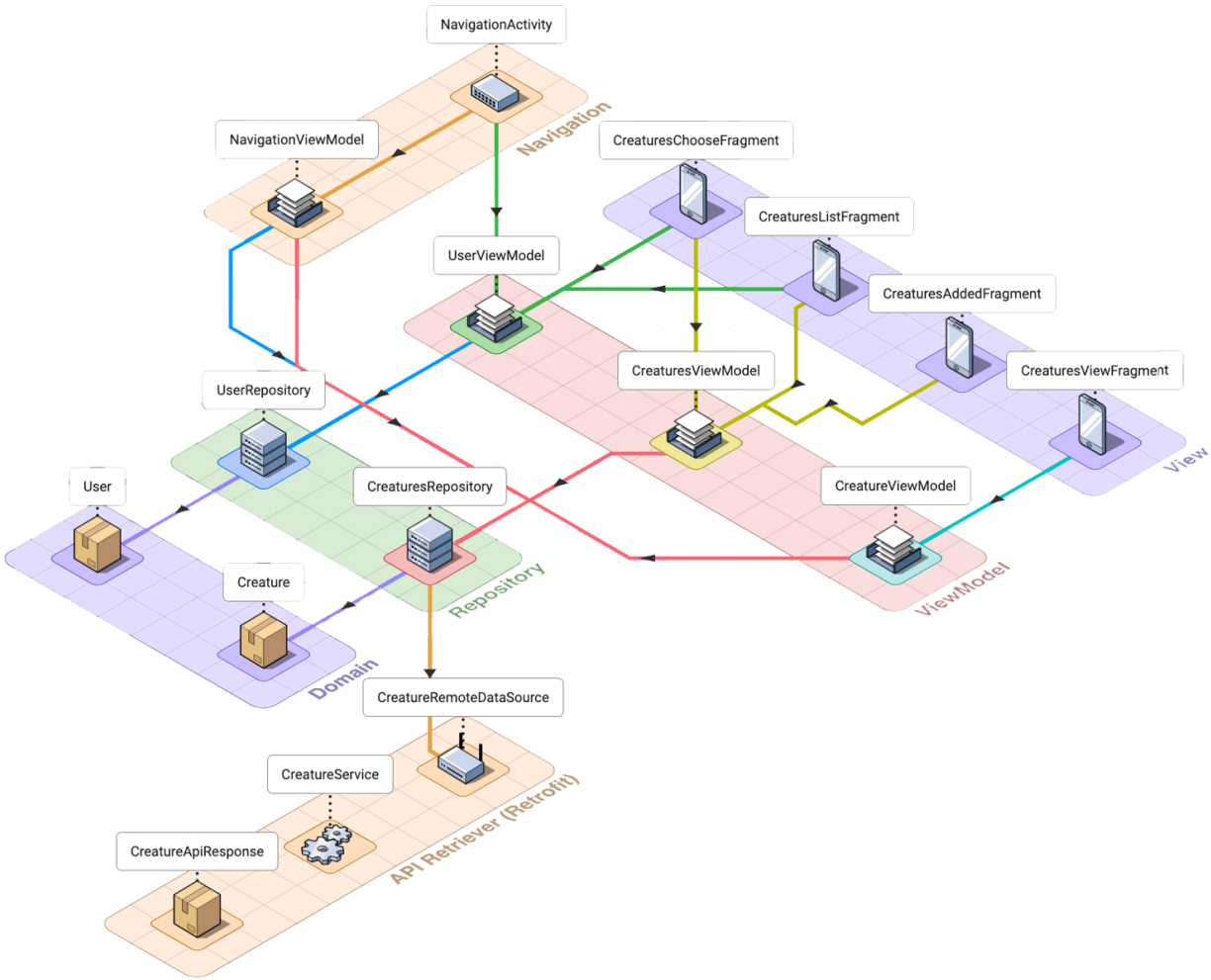


Figura 2.3: Diagrama da Arquitetura.

Bibliotecas utilizadas

Algumas implementações mais avançadas dariam muito trabalho para serem desenvolvidas sem o poder das bibliotecas! Por isso, utilizaremos algumas bibliotecas bem interessantes que ampliarão nossa capacidade de construção:

- **Navigation**: fluxo de navegação das telas;
- **Retrofit**: para requisições HTTP;
- **GSON**: para desserialização JSON;

- **Hilt**: para injeção de dependências;
- **RX**: para lidar com o fluxo de dados entre a camada *Model* e *View Model*;
- **Glide**: para renderização de imagens.

Trataremos dos detalhes de cada um desses conceitos e das bibliotecas mais à frente, além de importá-las conforme a necessidade.

Entidades do sistema

Como estamos desenvolvendo uma aplicação simples que permite possuir e cuidar de criaturas mágicas, teremos apenas duas entidades no sistema: **Creature** e **User**. Todo o funcionamento da aplicação girará em torno dessas duas entidades, que vão compor a camada de domínio (*Domain layer*) do software. É válido ressaltar que essa camada de domínio é considerada parte da camada *Model*, que detalharemos mais à frente.

Integração com back-end

Para obter a primeira lista de criaturas disponíveis, vamos utilizar um servidor back-end que possui um endpoint público para trazer essa lista. Trataremos do back-end mais à frente, quando estivermos construindo a lista de criaturas. O repositório do back-end também é público e é possível conferir o código-fonte em: https://github.com/FabricaDeSinapse/backend-livro-android-casa-do-codigo

Figura 2.4: Repositório do back-end no GitHub.

2.1 Arquitetura — View

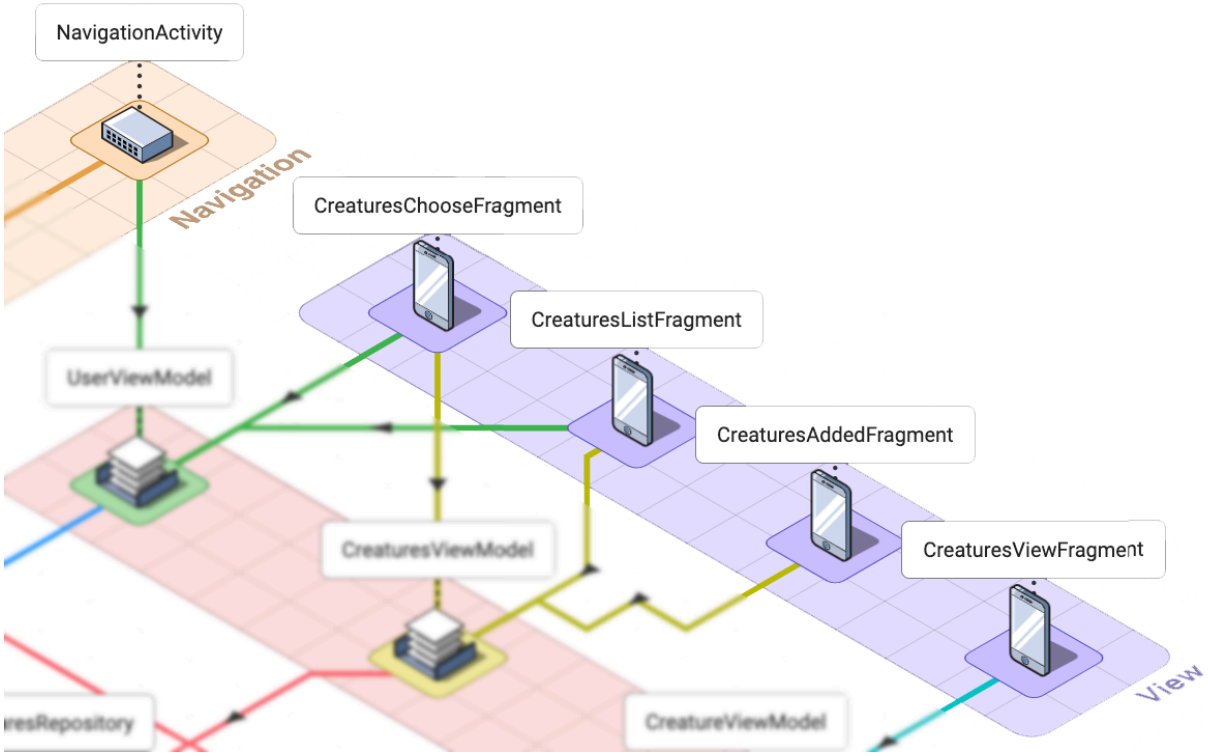


Figura 2.5: Arquitetura — View.

Para a estrutura da camada *View* do projeto, utilizaremos apenas uma única `Activity`, a `NavigationActivity`. Além da `Activity`, como é possível notar na imagem anterior, a aplicação possui as seguintes telas:

- `CreaturesChooseFragment`: obter uma criatura;
- `CreaturesListFragment`: lista de criaturas;
- `CreaturesAddedFragment`: criatura adicionada;
- `CreaturesViewFragment`: visualizar uma criatura.

Para construir o fluxo de navegação entre as telas do aplicativo, utilizaremos a biblioteca `Navigations` do Jetpack, que nos permite referenciar apenas o `Fragment` a ser exibido, como no exemplo a seguir.

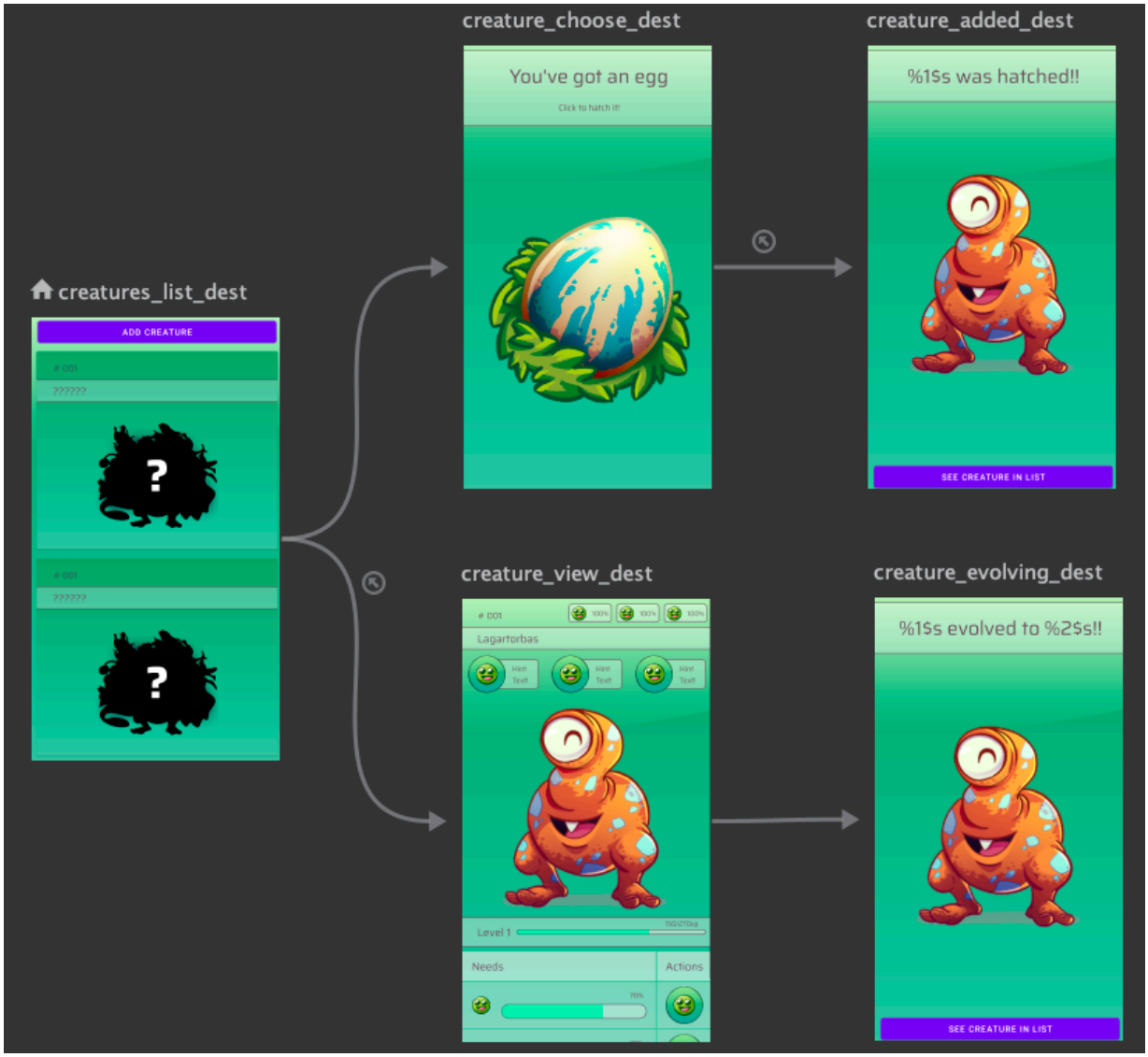


Figura 2.6: Fluxo de navegação das telas.

Trataremos dos detalhes da biblioteca `Navigation` mais à frente. Para alternar entre os `Fragments`, criaremos as `Actions`, que são ações que podem partir do usuário (através de um clique no botão, por exemplo) ou do próprio sistema (uma nova criatura foi recebida após um intervalo de tempo).

NOTA: durante a implementação do projeto deste livro, a tela de visualização da criatura é mais simples e não tem as funcionalidades de interação/evolução da criatura. Essas implementações fazem parte de um conteúdo extra e não são explicadas no livro, mas é possível acessá-las no repositório do projeto, no pacote **project-complete-with-extras**.

2.2 Arquitetura — ViewModel

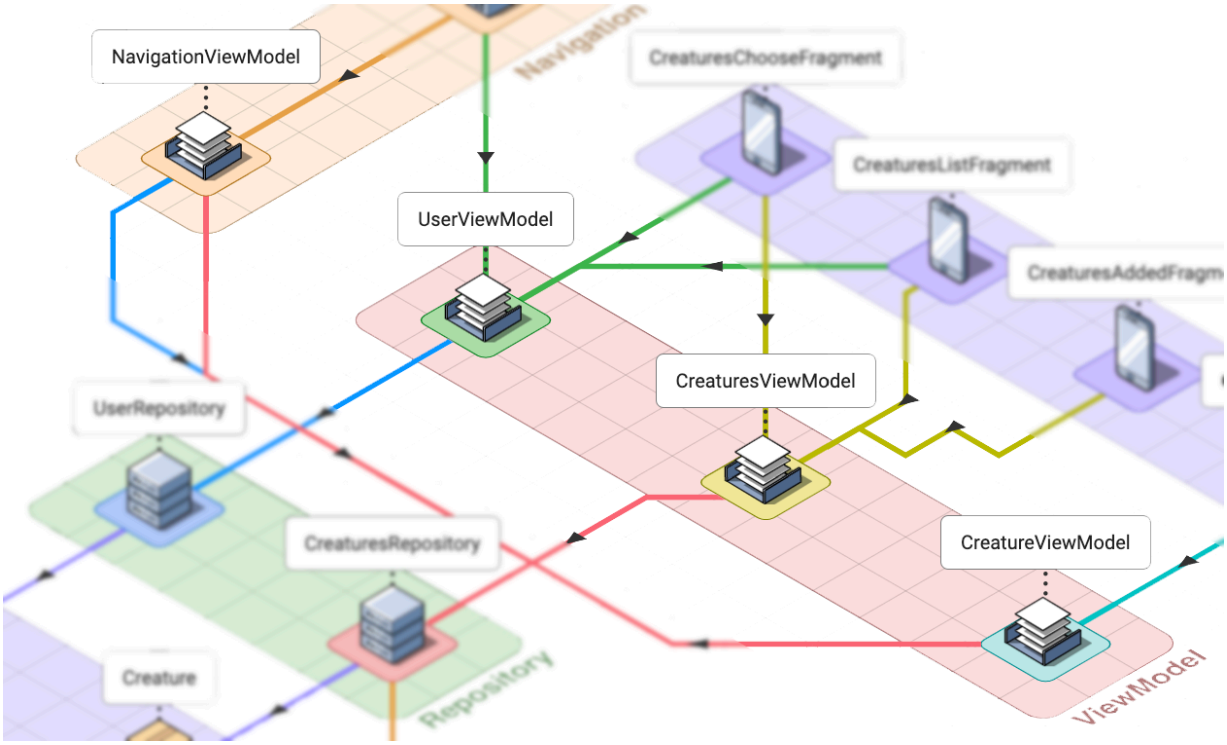


Figura 2.7: Arquitetura — ViewModel.

Como só temos duas entidades na aplicação, a estrutura da camada *View Model* é bem simples:

- `NavigationViewModel` : controla tudo o que está relacionado à navegação da aplicação. No caso, define quando os dados da

aplicação estão prontos para serem exibidos;

- `UserViewModel` : lida com todas as operações relacionadas a usuários;
- `CreaturesViewModel` : lida com todas as operações que tenham mais de uma criatura relacionada, como uma lista de criaturas disponíveis e a lista de criaturas possuídas pelo usuário;
- `CreatureViewModel` : lida com todas as operações individuais de uma criatura, como selecionar uma criatura e exibir as ações disponíveis para ela, que, no caso, são alimentar, treinar e brincar.

2.3 Arquitetura — Model

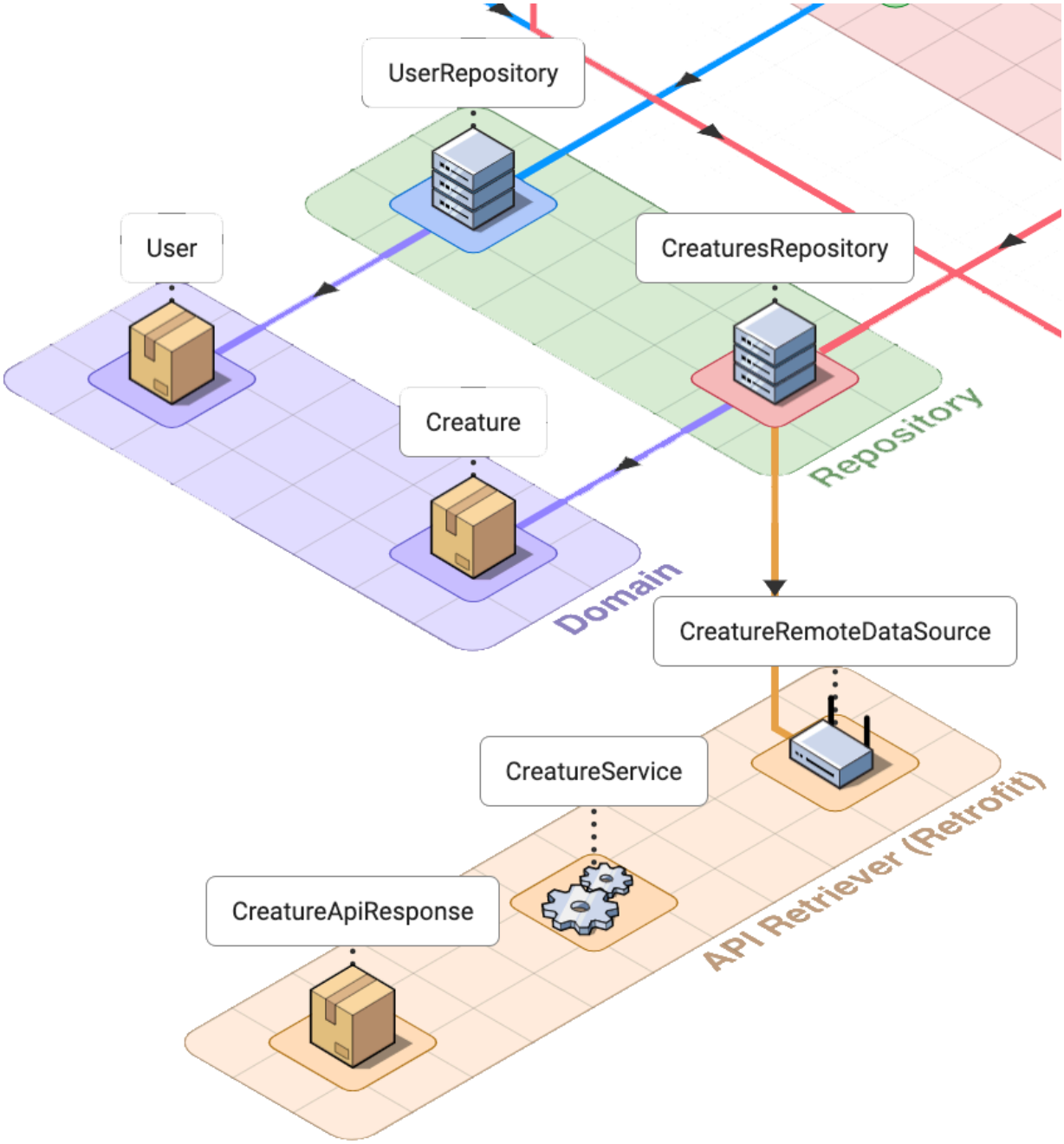


Figura 2.8: Arquitetura — Model.

A camada *Model* da aplicação contém toda a lógica de negócios e todo o funcionamento em si. Ela é responsável pelas requisições no back-end, por salvar as criaturas no banco de dados e atualizá-las quando as ações de alimentar, treinar ou brincar forem realizadas.

Um dos fatores mais importantes dessa camada é que todas as informações são expostas utilizando as entidades da camada de domínio, mencionada anteriormente. Dessa forma, as camadas *View* e *View Model* conseguem acessar as informações de uma forma simplificada, sem conhecer os detalhes de realizar uma requisição na API, por exemplo, separando as responsabilidades e deixando o código mais fácil de ler e de manter.

Para a *View*, não importa se a informação veio de um banco de dados no aparelho ou de um servidor na web; ela recebe esses dados da forma mais simples possível, seguindo a estrutura que foi definida na entidade de domínio, que, como o próprio nome diz, domina a aplicação como um todo.

Note também que há um item bem importante e que tem o papel de buscar informações em uma API na web. Essa implementação utilizada é o **Retrofit**, uma biblioteca da Square que permite realizar requisições HTTP para servidores na web. No nosso caso, pegaremos uma lista de criaturas que contém as informações básicas e a imagem de cada uma.

Dentro dessa camada, temos dois conceitos importantes: o ***Repository*** e o ***DataSource***. As classes que possuem o papel de ***Repository*** são o único ponto de entrada da camada *Model*. A responsabilidade delas é deixar a informação preparada para ser utilizada, independente se essa informação está armazenada no dispositivo ou em um servidor na web. Já as classes que possuem o papel de ***DataSource*** têm a responsabilidade de interagir com as fontes de dados.

Para dados armazenados no dispositivo, utilizaremos os ***LocalDataSources***, pois a palavra *local* é utilizada quando a

conexão é feita internamente. Quando esses dados estão armazenados em um servidor na web, utilizaremos os ***RemoteDataSources***, pois a palavra *remote* é utilizada quando estamos realizando uma conexão remota com algum dispositivo externo.

Próximos passos

Agora que entendemos a arquitetura do projeto de uma forma ampla, podemos criar o projeto e as primeiras classes. Seguiremos um fluxo mais natural de construção da aplicação, implementando algumas funcionalidades conforme a demanda. Ao final deste livro, teremos a mesma arquitetura da aplicação que foi apresentada neste capítulo.

Por último, mas não menos importante, o DevMon #2 se chama **Kotlin**, em homenagem à linguagem de programação que está ganhando cada vez mais espaço na comunidade Android e, também, na comunidade Java em geral, principalmente na área de back-end, seja Spring ou Ktor.

Por sua interoperabilidade com Java, a adesão ao Kotlin não é uma decisão muito difícil — tudo isso somado ao fato de que a empresa que criou o Kotlin, a JetBrains, possui IDEs muito poderosas como o Android Studio, IntelliJ IDEA, PyCharm, PHPStorm, entre outras. O IntelliJ IDEA e o Android Studio, por exemplo, possuem um kit de ferramentas muito bom para Kotlin, que converte códigos inteiros escritos em Java com apenas um atalho: `Ctrl + Alt + Shift + K`.

URL da imagem: https://i.imgur.com/EZ3zCBp.png.

Figura 2.9: O DevMon #2 se chama Kotlin!

Capítulo 3
Criando o projeto

Durante o desenvolvimento do livro, utilizaremos um repositório do GitHub que possui toda a referência do código que vamos escrever.

Cada capítulo do livro contém um determinado assunto ou uma variedade de assuntos que são demonstrados dentro de um projeto completo. O código de cada etapa está disponível em módulos separados, além de um módulo inicial, chamado `my-project`, que você pode usar para desenvolver o seu projeto, como é possível ver na imagem a seguir.

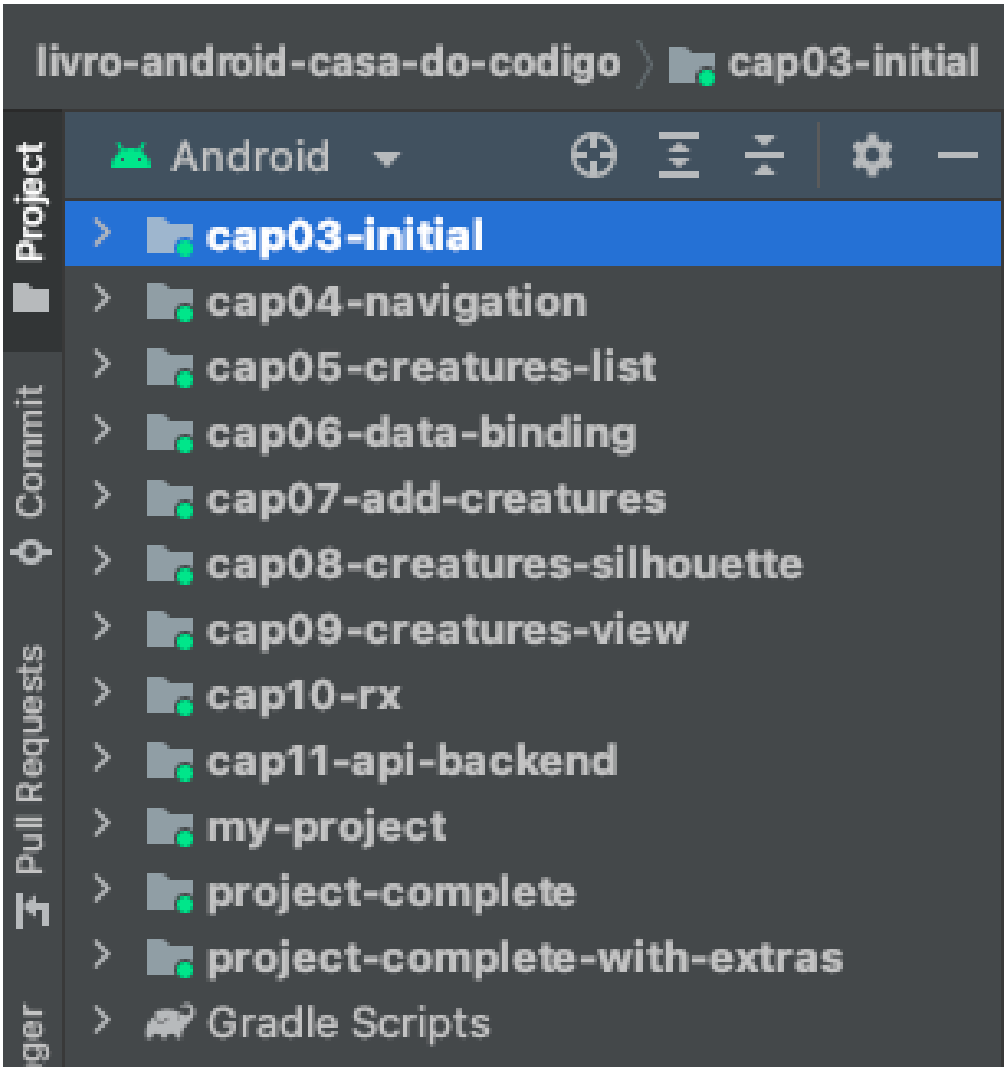


Figura 3.1: Visão dos módulos.

> Caso queira visualizar as implementações completas deste capítulo, acesse o módulo `CAP03-INITIAL`.

Encontrou algum problema?

Antes de começarmos a implementar códigos no projeto, é importante mencionar que o mundo da programação, principalmente do Android, evolui muito rápido e algumas implementações mudam ou deixam de funcionar. O intuito deste projeto, tanto do livro quanto do repositório, é que ele se mantenha atualizado pelo

máximo de tempo possível. Para isso, contamos com todo o apoio da comunidade e, principalmente, dos leitores e leitoras deste livro. Quaisquer problemas que você encontre, é possível reportá-los através do próprio GitHub, na aba de ***Issues***, em: https://github.com/FabricaDeSinapse/livro-android-casa-do-codigo/issues.

Figura 3.2: Página de issues do GitHub, para reportar problemas.

Os problemas reportados serão corrigidos tanto no repositório do GitHub quanto no livro. A versão mais atualizada do livro sempre estará disponível para download.

Playlist de vídeos

Este livro também conta com uma série de vídeos que mostram a construção do projeto, além de pontuar algumas explicações. Se você prefere aprender alguns conteúdos nesse formato, fique à vontade para conferir os vídeos, conforme evolui pelo livro.

https://youtube.com/playlist?list=PLzFYWfRVHp5JdMnr81O023vAbFb1h11GV.

Figura 3.3: Playlist de vídeos - canal do YouTube.

Para iniciar este capítulo que finalmente cria o projeto, vamos revelar qual é o DevMon deste capítulo. Após conhecermos as criaturas 1 e 2, Java e Kotlin, a criatura 3 corresponde à terceira evolução. No entanto, só revelaremos mais detalhes sobre ela mais à frente. Neste capítulo, vamos revelar o DevMon número 4. Seu nome é uma homenagem a uma linguagem de programação que é considerada a mãe de diversos sistemas e até mesmo de outras linguagens. Foi com essa linguagem de programação que o sistema operacional Android foi construído.

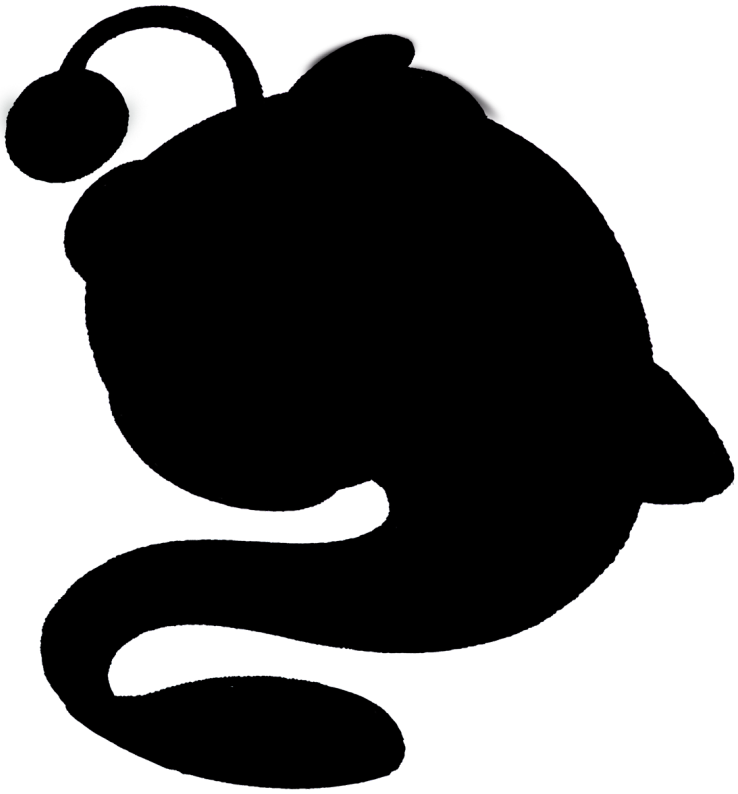


Figura 3.4: Qual é o DevMon #4?

3.1 Clonando o repositório

O primeiro passo para iniciar é clonar o repositório, através do link a seguir: https://github.com/FabricaDeSinapse/livro-android-casa-do-codigo.

Figura 3.5: Repositório no GitHub.

Certifique-se de que você possui o software Git instalado ou alguma alternativa como o GitHub Desktop: https://git-scm.com/downloads

Figura 3.6: Download do Git.

Com o Git instalado, abra qualquer terminal e digite o comando a seguir para fazer o clone:

```
git clone https://github.com/FabricaDeSinapse/livro-android-casa-do-codigo.git
```

Se preferir, você pode fazer o clone utilizando o SSH, com o comando a seguir:

```
git clone git@github.com:FabricaDeSinapse/livro-android-casa-do-codigo.git
```

Com o repositório clonado, abra-o com o Android Studio, de preferência na versão mais atualizada. Abra o módulo `my-project` e navegue até o pacote principal da aplicação, dentro da pasta `java`. Você encontrará a `Activity` principal do projeto, chamada `NavigationActivity`. Como mencionado anteriormente, este projeto utilizará apenas uma única `Activity`, que será utilizada em conjunto com a biblioteca `Navigations`, do Android Jetpack. Todas as telas da aplicação serão construídas a partir de `Fragments`, que vamos criar e evoluir ao longo dos próximos capítulos.

Além disso, no arquivo `build.gradle` do projeto, que é possível encontrar dentro da aba `Gradle Scripts`, controlaremos todas as versões das bibliotecas necessárias para o projeto. Dessa forma, garantimos que todos os módulos possuirão as dependências sempre atualizadas.

3.2 Injeção de dependência (DI)

Apesar de ser algo novo para muitos devs, esse assunto está presente desde o começo dos anos 1990, com abordagens em diversos livros e para diversas linguagens de programação, e há uma série de bibliotecas capazes de facilitar sua utilização.

Esse conceito visa reduzir o nível de acoplamento entre diferentes módulos de um sistema, ou seja, em vez de um módulo que tem uma dependência ser o responsável por satisfazer essa necessidade, isso é feito pela configuração do software, que vai "injetar" em cada

componente as dependências que foram declaradas. Essa prática está relacionada com o padrão *Inversion of Control* (inversão de controle), que consiste em definir quem tem o controle das instâncias de uma aplicação, geralmente delegando a responsabilidade de construção de novas instâncias para uma classe conhecida como *Factory*, que, além de construí-las, também as reutiliza conforme a necessidade.

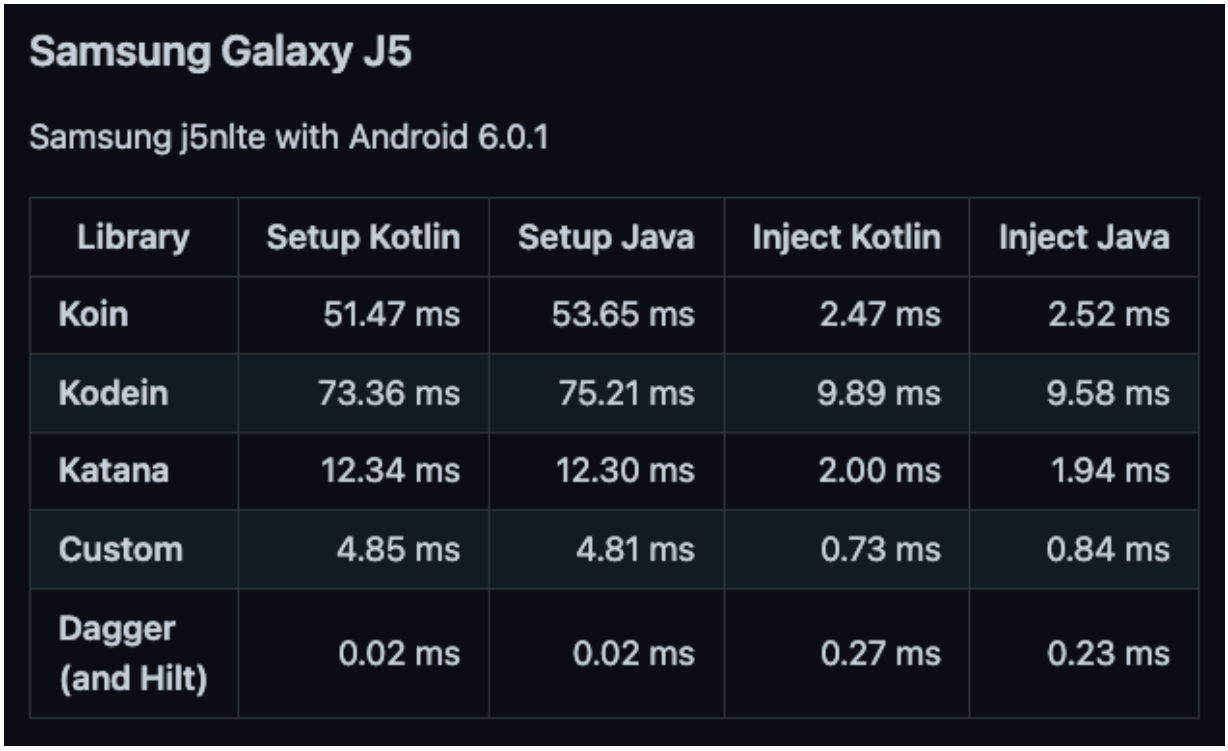
A injeção de dependência (ou DI, do inglês *Dependency Injection*) é uma técnica de programação que pode ser realizada em qualquer linguagem de programação, e de várias maneiras. Em qualquer projeto, seja ele grande ou pequeno, mesmo que nós não tenhamos percebido, existem casos em que objetos dependem de objetos; ou seja, para construir uma nova informação, precisamos que outra informação também seja construída.

3.3 Bibliotecas para DI

Existem diversas bibliotecas disponíveis para tratar a injeção de dependência, independente da linguagem de programação ou do framework em que estamos trabalhando. No Java, temos algumas das principais bibliotecas que evoluíram ao longo do tempo, melhorando cada vez mais a utilização e, principalmente, a performance. No ecossistema Android, as principais bibliotecas utilizadas são **Guice**, **Dagger v1**, **Dagger v2**, **Koin** e **Hilt**.

Como podemos notar, injeção de dependências é um assunto com várias opções e diferentes propostas de solução. Atualmente, muitas aplicações Android acabam optando por Dagger 2, Koin ou Hilt, levando os diferentes pontos mencionados em consideração. Por conta principalmente dos fatores **checagem de erros em tempo de compilação** e **performance da aplicação** (como é possível ver na imagem a seguir), utilizaremos a biblioteca **Hilt** neste livro,

que nada mais é do que a Dagger 2 por baixo dos panos, com uma configuração simplificada para aplicações Android.



Samsung Galaxy J5

Samsung j5nlte with Android 6.0.1

Library	Setup Kotlin	Setup Java	Inject Kotlin	Inject Java
Koin	51.47 ms	53.65 ms	2.47 ms	2.52 ms
Kodein	73.36 ms	75.21 ms	9.89 ms	9.58 ms
Katana	12.34 ms	12.30 ms	2.00 ms	1.94 ms
Custom	4.85 ms	4.81 ms	0.73 ms	0.84 ms
Dagger (and Hilt)	0.02 ms	0.02 ms	0.27 ms	0.23 ms

Figura 3.7: Comparação de performance das bibliotecas.

3.4 Implementando o Hilt no projeto

Para implementar o Hilt no nosso projeto Android, precisamos importar a biblioteca em dois lugares: no `build.gradle` do projeto e, depois, no `build.gradle` da aplicação. Como nesse projeto de exemplo temos vários `build.gradle` — um para o seu projeto, um para o projeto completo e outro para cada capítulo deste livro —, certifique-se de selecionar o arquivo `build.gradle` do `my-project`.

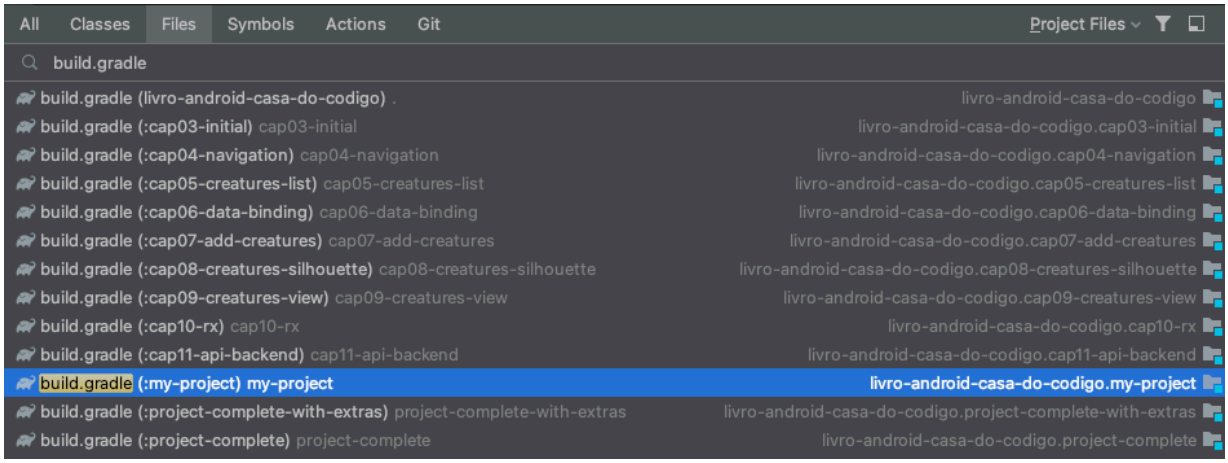


Figura 3.8: Arquivo build.gradle do app my-project.

Abra o `build.gradle` do `my-project` e adicione o *import* dos plug-ins Kapt e Hilt, além das dependências do ViewModel, LiveData e Navigation:

```
plugins {
    id 'com.android.application'
    id 'kotlin-android'

    // Kapt & Hilt
    id "kotlin-kapt"
    id "dagger.hilt.android.plugin"
}

//...
dependencies {
    //...

    // Hilt
    implementation("com.google.dagger:hilt-android:$hiltVersion")
    kapt("com.google.dagger:hilt-android-compiler:$hiltVersion")

    // ViewModel & LiveData
    implementation("androidx.lifecycle:lifecycle-livedata-
ktx:$lifecycleVersion")
    implementation("androidx.lifecycle:lifecycle-viewmodel-
ktx:$lifecycleVersion")
```

```
    implementation("androidx.hilt:hilt-lifecycle-
viewmodel:$hiltVMVersion")

    // Navigation
    implementation("androidx.navigation:navigation-fragment-
ktx:$navigationVersion")
    implementation("androidx.navigation:navigation-ui-
ktx:$navigationVersion")
}
```

Como o Dagger é um gerador de código em tempo de compilação (e, portanto, o Hilt também), é necessário um *annotation processor* para isso. No caso do Kotlin, utilizamos o `kotlin-kapt`. Além disso, precisamos registrar o Hilt como plug-in e importar sua dependência comum, via `implementation` e sua dependência do *kapt*. Por fim, como utilizaremos o Hilt em conjunto com o `ViewModel` do Android Jetpack, adicionamos uma extensão específica para isso.

A dependência do Navigation serve para criarmos o fluxo de navegação da aplicação, que será feito no próximo capítulo. Além disso, ela traz a declaração do `by viewModels()` que serve facilitar o processo de instanciar um `ViewModel`, que faremos à frente.

Também precisamos garantir que o Hilt esteja habilitado como um plug-in do *Gradle*, direto no arquivo `build.gradle` do projeto todo. Para abrir esse arquivo, certifique-se de que selecionou o `build.gradle` com o nome `livro-android-casa-do-codigo` ao lado.

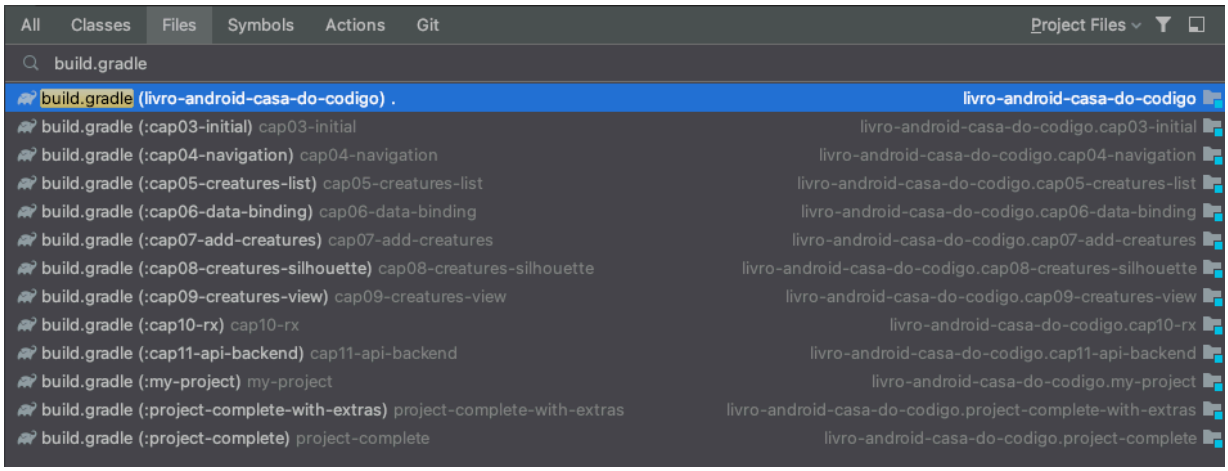


Figura 3.9: Arquivo build.gradle do projeto.

> Note que esse arquivo começa com várias declarações de versões de bibliotecas e contém algumas dependências já referenciadas.

```
// Top-level build file where you can add configuration options common to
all sub-projects/modules.
buildscript {
    ext {
        // Core
        coreKtxVersion = "1.7.0"
        appCompatVersion = "1.4.1"

        // Layout
        materialVersion = "1.5.0"
        constraintLayoutVersion = "2.1.3"
        legacySupportVersion = "1.0.0"

        // ...
```

Ao longo dos próximos capítulos, vamos utilizar várias bibliotecas, que são atualizadas constantemente. Para manter o livro em dia com as bibliotecas, utilizaremos esse único arquivo `build.gradle` para gerenciar todas as dependências, de todos os capítulos do livro.

Portanto, sempre que for necessário importar uma biblioteca nova, as versões estarão centralizadas em um único local.

> Sempre que mudar algum código no `build.gradle`, aparecerá um botão com a opção de `Sync now` para aplicar as atualizações. Clique no botão para garantir que o Gradle fará o trabalho de importar tudo.

Com o Hilt importado, iniciamos a configuração criando uma classe que estenda a classe `Application`, do Android, definindo o ponto de partida de todas as injeções. Essa classe deverá ser anotada com `@HiltAndroidApp`.

Crie um arquivo chamado `App.kt` no pacote principal da aplicação e adicione o seguinte conteúdo:

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore

import android.app.Application
import dagger.hilt.android.HiltAndroidApp

@HiltAndroidApp
class App : Application()
```

Para que o Android inicie o app a partir da classe que acabamos de criar, precisamos referenciá-la no `AndroidManifest.xml`, da seguinte forma:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="tech.salvatore.livro_android_kotlin_paulo_salvatore">

    <application
        android:name=".App"
// ...
```

Pronto! Agora o Hilt está configurado para a aplicação. Rode o projeto para garantir que a configuração inicial foi realizada com

sucesso.

Com a aplicação configurada, precisamos marcar as `Activities` para poderem receber dependências de outros lugares, afinal, é por elas que todo app Android começa. Faremos isso anotando a `NavigationActivity` com `@AndroidEntryPoint`:

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import dagger.hilt.android.AndroidEntryPoint

@AndroidEntryPoint
class NavigationActivity : AppCompatActivity() {
// ...
```

Com isso, se precisarmos conectar a `Activity` com alguma outra classe do sistema, ela está preparada para que isso seja feito.

Lembra quando falamos sobre a arquitetura do projeto e mencionamos a arquitetura MVVM (*Model*, *View* e *View Model*)? A `Activity` faz parte da *View* e costuma buscar todos os dados da aplicação na camada de *View Model*; portanto, toda vez que precisarmos conversar com uma dessas classes da camada *View Model*, utilizaremos o Hilt para fornecer e gerenciar a construção delas.

Para exemplificar, vamos criar um `ViewModel` para cuidar da tela de navegação. Crie um novo arquivo no pacote principal da aplicação, do tipo `Kotlin File/Class` e com o nome `NavigationViewModel.kt`. Dentro desse arquivo, crie uma classe com o mesmo nome, chamada `NavigationViewModel`. Para garantir que o Jetpack entre em ação, vamos conectá-la com a classe `ViewModel` dentro do Jetpack através do pacote *Lifecycle*.

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore

import androidx.lifecycle.ViewModel
```

```
class NavigationViewModel : ViewModel() {
}
```

Agora, para que o Hilt conheça esse *View Model*, anotaremos com `@HiltViewModel`, da seguinte forma:

```
import dagger.hilt.android.lifecycle.HiltViewModel

@HiltViewModel
class NavigationViewModel : ViewModel() {
```

Como o Hilt depende da *annotation* `@Inject()` para saber como injetar uma determinada classe, precisamos adicioná-la no construtor do `NavigationViewModel`:

```
import javax.inject.Inject

@HiltViewModel
class NavigationViewModel @Inject constructor() : ViewModel()
```

Para receber uma instância nova do `NavigationViewModel` dentro da `NavigationActivity`, basta utilizar a declaração `by viewModels()`:

```
import androidx.activity.viewModels

@AndroidEntryPoint
class NavigationActivity : AppCompatActivity() {
    private val navigationViewModel: NavigationViewModel by viewModels()
```

Com isso, o `NavigationViewModel` estará disponível no momento em que alguém precisar usá-lo pela primeira vez.

Além disso, caso essa `Activity` seja destruída permanentemente, o `ViewModel` será destruído junto. Nos próximos capítulos, trataremos um pouco mais dos detalhes do `ViewModel` e como utilizá-lo na aplicação.

Conclusão

Terminamos a etapa inicial do nosso projeto, que prepara uma arquitetura bem robusta de início, configurando todo o mecanismo de injeção de dependência e preparando a única `Activity` da aplicação, que será responsável por toda a navegação dentro do app.

Não se esqueça de realizar o *commit* do que fizemos até aqui em algum repositório seu no GitHub. O repositório pode ser público e você pode usar esse projeto em seu portfólio tranquilamente, pois todos os códigos estão sob a licença *open source* MIT License.

> Saiba mais sobre a licença no arquivo `LICENSE.md`, na raiz do projeto, ou através do link público no GitHub: https://github.com/FabricaDeSinapse/livro-android-casa-do-codigo/blob/main/LICENSE.

No próximo capítulo, implementaremos o fluxo de navegação das telas, criaremos os primeiros `Fragments` com os *layouts* iniciais e veremos a nossa primeira criaturinha direto no app!

Por fim, chegou a hora de revelar que o nome do DevMon #4 foi uma homenagem à linguagem de programação **C**. Como mencionado anteriormente, por conta da velocidade e do desempenho necessários em um sistema operacional, o kernel (núcleo) do sistema utiliza duas linguagens: C e Assembly. Os sistemas mais famosos, como Windows, Linux, MacOS e Android, utilizam essas linguagens de programação em suas bases.

URL da imagem: https://i.imgur.com/iWeXxA9.png.

Figura 3.10: O DevMon #4 se chama C!

Leitura complementar: como funciona a DI?

Caso o conceito de injeção de dependência não esteja muito claro, a leitura deste tópico facilitará entender o problema que estamos resolvendo e o principal conceito aplicado pelas bibliotecas de DI.

Visualizar a aplicação desse conceito na prática facilita muito o entendimento. Vamos dar uma olhada em como o fluxo de injeção das dependências acontece na aplicação que estamos construindo. O diagrama inicial da aplicação pode ser observado na figura a seguir.

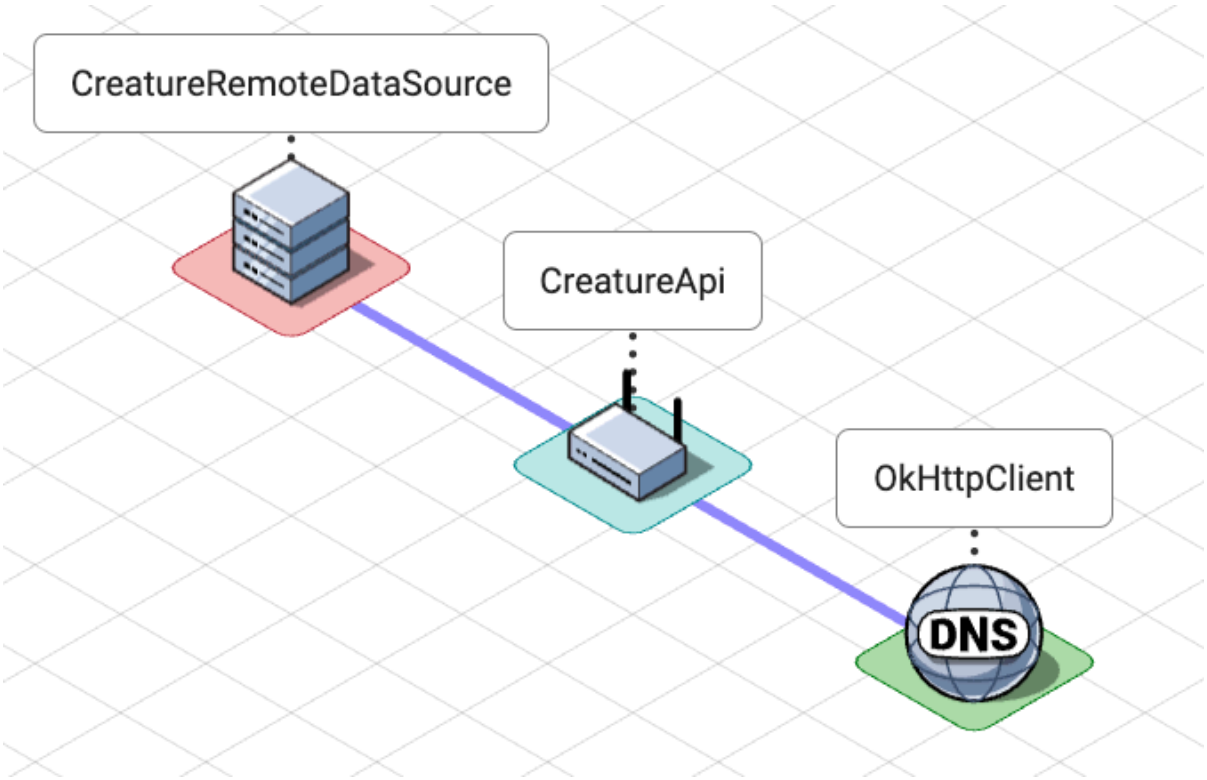


Figura 3.11: Diagrama inicial da aplicação.

Antes de aprender sobre as bibliotecas, vamos entender exatamente o que é a injeção de dependência e qual problema ela resolve. Analisando o funcionamento da aplicação, a primeira tela exibe uma lista de criaturas. No entanto, para saber quais são as criaturas e suas informações, é necessário realizar uma solicitação a um servidor que está na internet. Para isso, vamos preparar uma classe que é responsável por acessar esse servidor, chamada `CreatureRemoteDataSource`, que possui o método `getCreatures()`, que faz a requisição e retorna a lista de criaturas.

```
class CreatureRemoteDataSource {
  fun getCreatures(): List<Creature> {
     val api = CreatureApi()
     return api.findAll()
  }
}
```

Perceba que, nesse caso, a classe `CreatureRemoteDataSource` tem a sua primeira dependência, a classe `CreatureApi`.

Como a construção da `CreatureApi` está sendo feita dentro da classe `CreatureRemoteDataSource`, criamos um débito técnico que podemos pagar no futuro, pois qualquer alteração no construtor da classe `CreatureApi` deverá ser atualizada nesse local também. A princípio, pode não parecer um problema tão grande, mas imagine que a classe `CreatureApi` seja utilizada em 10 outros trechos — aí, sim, começamos a ter um problema maior. Mas, por enquanto, vamos deixar esse problema de lado e evoluir um pouco mais a implementação.

Como vimos no exemplo anterior, temos `getCreatures()`, de `CreatureRemoteDataSource`, executando `findAll()`, de `CreatureApi`. Para que o `findAll()` consiga retornar a lista de criaturas, é necessário acessar um servidor que está na internet através do protocolo HTTP. Para isso, vamos exemplificar com a biblioteca `OkHttpClient`, da Square.

Assim que o `findAll()` for executado, ele precisa obter uma instância de `OkHttpClient`, realizando a implementação de acordo com a documentação da biblioteca. O exemplo a seguir é ilustrativo, mas deve servir para exemplificar uma requisição sendo feita e retornando a lista de criaturas.

```
class CreatureApi {
  fun findAll(): List<Creature> {
     val client = OkHttpClient()
     val request =
Request.Builder().url("https://api.devmon.com/creature").build()
     return client.newCall(request).execute()
  }
}
```

Pronto! Agora que temos tudo declarado, podemos realizar a execução dos métodos e receber a lista atualizada de criaturas. Para

ilustrar, faremos isso em um objeto chamado `Program`, que tem um método principal chamado `main()`, como no trecho a seguir.

```
object Program {
  fun main() {
    val remoteDataSource = CreatureRemoteDataSource()
    val creatures = remoteDataSource.getCreatures()
    println(creatures)
  }
}
```

Com isso, nosso primeiro exemplo chega ao fim. Temos o objeto `Program` criando a instância `CreatureRemoteDataSource` e solicitando a lista de criaturas. Quando essa lista é solicitada, o método `getCreatures()`, da classe `CreatureRemoteDataSource`, cria a instância de `CreatureApi`, executando o método `findAll()`. Por fim, o método `findAll()` cria a instância de `OkHttpClient()` e realiza a requisição HTTP ao servidor.

Evoluindo a aplicação

Conforme o tempo passa, novas requisições vão sendo necessárias e nossa aplicação começa a crescer. Em algum momento, vamos precisar implementar um tratamento de erro que sirva para todas as requisições ou alguma outra implementação que precisamos para ter controle da instância do `OkHttpClient`, por exemplo.

Quando esse dia chegar, será necessário procurar por todos os métodos que criam instâncias para realizar a atualização em cada um deles. Nesse momento, teremos que pagar o débito técnico que foi criado no momento em que esse padrão de código começou a ser utilizado. Para evitar que isso ocorra, é preciso mudar quem tem controle da criação das dependências, em vez de cada um criar sua própria instância.

Existem algumas formas de fazer isso. Nesse momento, trataremos apenas uma delas, que consiste em não se preocupar com a criação da instância da classe que estamos chamando. Em vez disso,

declaramos no construtor qual é a classe necessária para garantir o funcionamento, da seguinte forma:

```
class CreatureApi(private val client: OkHttpClient) {
  fun findAll(): List<Creature> {
    // ...
```

Dessa forma, a responsabilidade de fornecer a instância foi transferida para quem estiver utilizando a classe `CreatureApi`, que, no caso, é a classe `CreatureRemoteDataSource`.

Agora que temos as dependências da `CreatureApi` declaradas, precisamos satisfazê-las na classe `CreatureRemoteDataSource`. Antes disso, vamos dar uma olhada no código.

```
class CreatureRemoteDataSource {
  fun getCreatures(): List<Creature> {
     val api = CreatureApi()
     return api.findAll()
  }
}
```

Como acabamos de adicionar um parâmetro no construtor `CreatureApi`, precisamos adaptar a construção que está sendo feita dentro de `getCreatures()`. Para corrigir o problema, podemos fornecer uma instância de `OkHttpClient` ou mover a criação da instância de `CreatureApi` para o construtor da classe `CreatureRemoteDataSource`.

1. Se criarmos a instância de `OkHttpClient` dentro do método `getCreatures()` e fornecê-la para a `CreatureApi` que está sendo criada, resolve o problema para a criação de instância `CreatureApi`, mas traz uma nova responsabilidade para a classe `CreatureRemoteDataSource`, que é a de gerenciar a criação da instância de `OkHttpClient`;
2. Ao mover a dependência de `CreatureApi` para o construtor `CreatureRemoteDataSource`, estamos tirando a responsabilidade da

classe de criar essa instância, fazendo com que quem a utilize seja responsável por gerenciar e informar essa instância.

Diante desse cenário, vamos resolver da mesma forma que fizemos antes: transferir a responsabilidade da instância da `CreatureApi` para quem estiver precisando da classe `CreatureRemoteDataSource`, movendo a declaração para o construtor.

```
class CreatureRemoteDataSource(private val api: CreatureApi) {
  fun getCreatures(): List<Creature> {
     return api.findAll()
  }
}
```

Com essa implementação realizada, podemos adaptar o código do objeto `Program` para construir todas as instâncias necessárias, da seguinte forma:

```
object Program {
  fun main() {
    val client = OkHttpClient()
    val creatureApi = CreatureApi(client)
    val remoteDataSource = CreatureRemoteDataSource(creatureApi)
    val creatures = remoteDataSource.getCreatures()
    println(creatures)
  }
}
```

Pronto! Com isso, nós realizamos nossa primeira injeção de dependências. Podemos ver que, na prática, tudo é resolvido apenas com argumentos nos construtores das classes, eliminando a responsabilidade da classe de realizar internamente a construção de uma de suas dependências e possibilitando que as instâncias sejam manipuladas e reutilizadas em níveis mais superiores do código.

Fábricas de instâncias

No exemplo que construímos, todas as instâncias são criadas e gerenciadas pelo objeto `Program`. No entanto, na vida real não é tão

simples assim, uma vez que um aplicativo Android, por exemplo, possui várias telas que devem ser inicializadas — na maioria dos casos, de forma independente umas das outras. Por conta disso, delegar toda a responsabilidade de gerenciamento das instâncias para um único lugar é um cenário utópico, o que nos traz um novo conceito: a **fábrica de instâncias**.

Como o próprio nome diz, a responsabilidade dessa classe é a de fabricar instâncias de classes específicas, concentrando toda a construção de cada classe em uma fábrica que tem essa única responsabilidade. Para implementá-la na classe `CreatureApi`, por exemplo, basta criar uma classe (no Kotlin, pode ser um `object`, que é uma classe com apenas uma única instância) chamada `CreatureApiFactory` que possui um método `getInstance()` que retorna uma instância de `CreatureApi` pronta para ser usada. Como sabemos, a classe `CreatureApi` depende da classe `OkHttpClient`. Nesses casos, a classe `Factory` deve chamar outras classes `Factory` para satisfazer todas as suas dependências, criando um ecossistema de fábricas que gerenciam suas respectivas classes, facilitando a manutenção do código.

Aplicando a fábrica de instâncias em nosso exemplo de código, deverá ficar como o código a seguir.

```
// Código da fábrica reutiliza as instâncias
object CreatureRemoteDataSourceFactory {
    private var instance: CreatureRemoteDataSource?

  fun getInstance(): CreatureRemoteDataSource {
    if (instance == null) {
      instance =
CreatureRemoteDataSource(CreatureApiFactory.getInstance())
    }

    return instance!!
  }
}
```

```
// Código das outras fábricas na mesma estrutura de
// `CreatureRemoteDataSourceFactory`, mudando apenas os tipos

object CreatureApiFactory { /* ... */ }
object OkHttpClientFactory { /* ... */ }

// Com as fábricas prontas, basta utilizá-las onde quiser

object Program {
  fun main() {
    val remoteDataSource = CreatureRemoteDataSourceFactory.getInstance()
    val creatures = remoteDataSource.getCreatures()
    println(creatures)
  }
}
```

Nesse exemplo, podemos perceber que a criação de uma fábrica é algo relativamente simples, mantendo uma estrutura muito semelhante e que reutiliza as instâncias, evitando um possível desperdício de memória na criação de muitas instâncias.

Apesar de as fábricas de instâncias serem simples de implementar e resolverem o problema em um primeiro momento, existem alguns problemas que podem aparecer com o tempo e tomarão tempo de desenvolvimento à medida que o projeto avança.

Um primeiro problema, muito comum, é a *reutilização de instâncias*. No exemplo anterior de código, a classe `CreatureRemoteDataSourceFactory` fornece sempre a mesma instância de `CreatureRemoteDataSource` — a instância só é criada assim que o método `getInstance()` for chamado pela primeira vez. Depois disso, a instância é armazenada em uma propriedade da classe e reutilizada sempre que alguém quiser uma nova instância. Dessa forma, a instância é compartilhada por toda a aplicação. Isso serve para a maioria dos casos, mas também é comum precisarmos de instâncias que sejam reutilizadas apenas internamente, entre camadas diferente da aplicação — por exemplo, ter uma instância de

`OkHttpClient` para cada classe de `Api`, como a `CreatureApi` e outras que podem existir.

Além do problema mencionado, gerenciar a *remoção de instâncias* também é algo a ser considerado, principalmente quando precisamos reduzir o consumo de memória do aplicativo. Por conta desses e de outros problemas, nem sempre utilizar as `Factories` pode ser um bom caminho, afinal, trará trabalho de código adicional, fora o tempo de planejamento para resolver cada um dos pontos.

Por conta disso, bibliotecas que resolvem o problema de injeção de dependências foram surgindo ao longo dos anos, começando por bibliotecas que resolvem o problema usando apenas Java, até bibliotecas específicas para Kotlin e Android.

CAPÍTULO 4
Fluxo de navegação

> Caso queira visualizar as implementações completas deste capítulo, acesse o módulo **CAP04-NAVIGATION**.

Como mencionado anteriormente, para construir o fluxo de navegação entre as telas do aplicativo, utilizaremos a biblioteca `Navigations` do Jetpack, que nos permite referenciar apenas o `Fragment` que será exibido no momento, de acordo com a imagem a seguir. Além da `NavigationActivity`, a aplicação possui as seguintes telas:

- `CreaturesListFragment` : ver lista de criaturas;
- `CreaturesChooseFragment` : obter uma criatura;
- `CreaturesAddedFragment` : ver criatura adicionada;
- `CreaturesViewFragment` : visualizar uma criatura;
- `CreaturesEvolvingFragment` : mostrar a evolução.

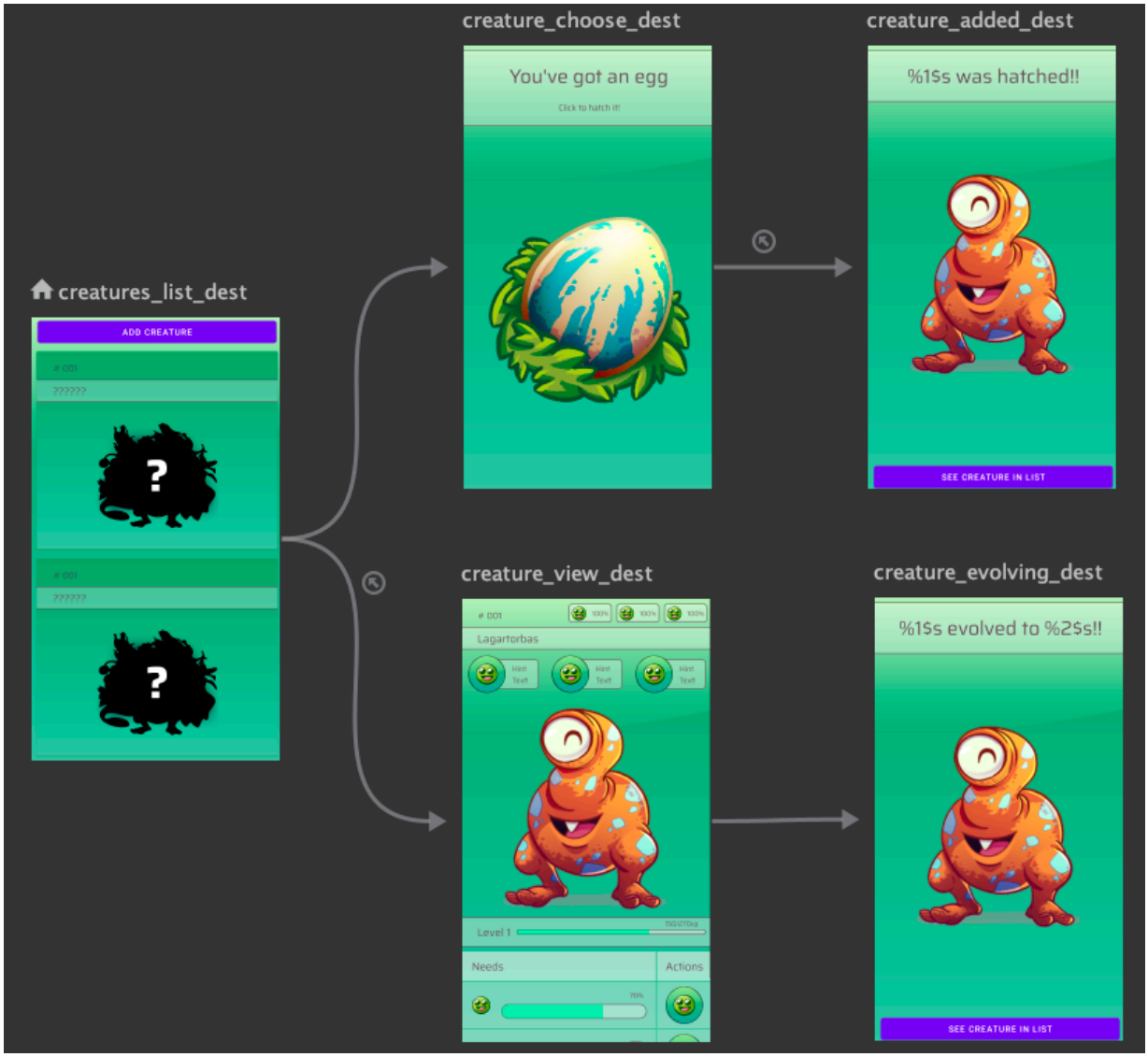


Figura 4.1: Fluxo de navegação completo da aplicação.

Mantendo a tradição do início dos capítulos, a quarta criatura que será revelada é o DevMon #5. Ela é a evolução da criatura C, que apresentamos no capítulo anterior. Ela também é nomeada em homenagem a uma linguagem de programação muito poderosa, que está presente no universo da tecnologia há muitos anos — desde 1979 — e que continua recebendo atualizações constantemente, sendo a linguagem mais popular comercialmente nos anos 1990 e bastante utilizada na academia.

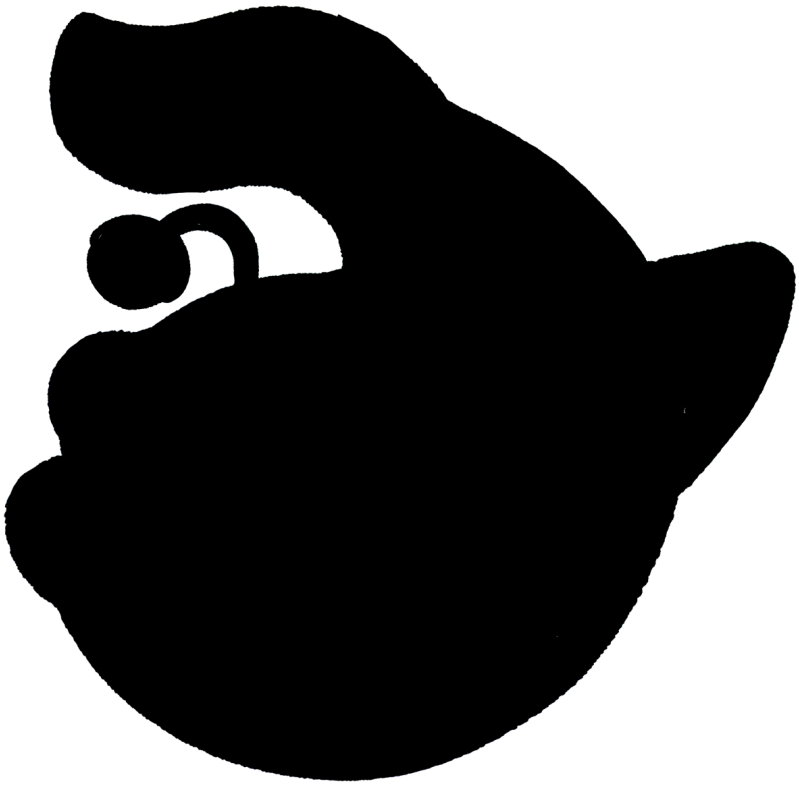


Figura 4.2: Qual é o DevMon #5?

4.1 Criando a lista de criaturas

Vamos começar pela criação do `CreaturesListFragment`, que exibirá a lista de criaturas. Clique com o botão direito no módulo do pacote principal da aplicação e selecione a opção `New | Kotlin Class/File`, como é possível ver na imagem a seguir.

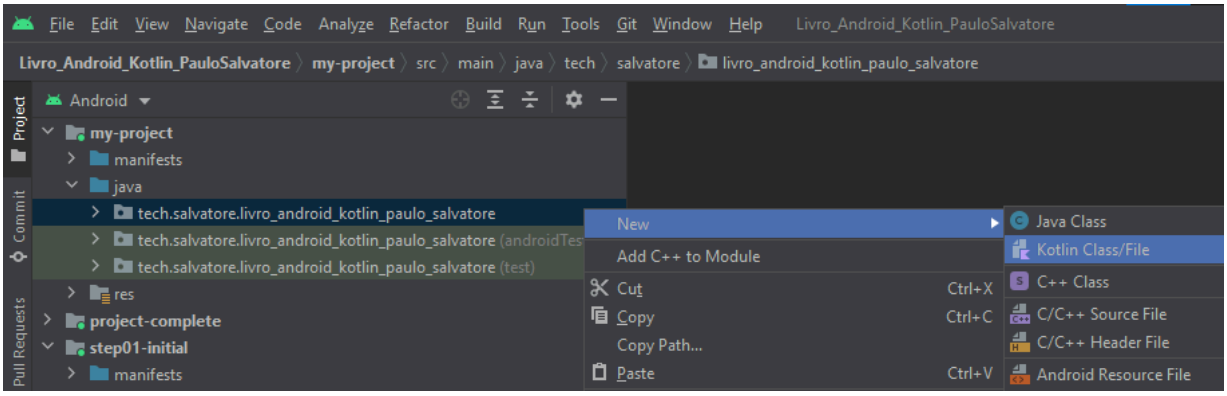


Figura 4.3: Criação do Fragment.

Nomeie o arquivo como `CreaturesListFragment` e pressione *enter*. Para transformar esse arquivo em um `Fragment` compatível com o Android, adicione o seguinte *boilerplate*:

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore

import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment

class CreaturesListFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        return inflater.inflate(R.layout.creatures_list_fragment,
container, false)
    }
}
```

Note que a declaração `R.layout.fragment_creatures_list` está com erro, pois precisamos criar o arquivo de *layout*. Para isso, navegue até a pasta `res/layout`, clique com o botão direito e selecione a opção `New | Layout Resource File`, como mostra a imagem a seguir.

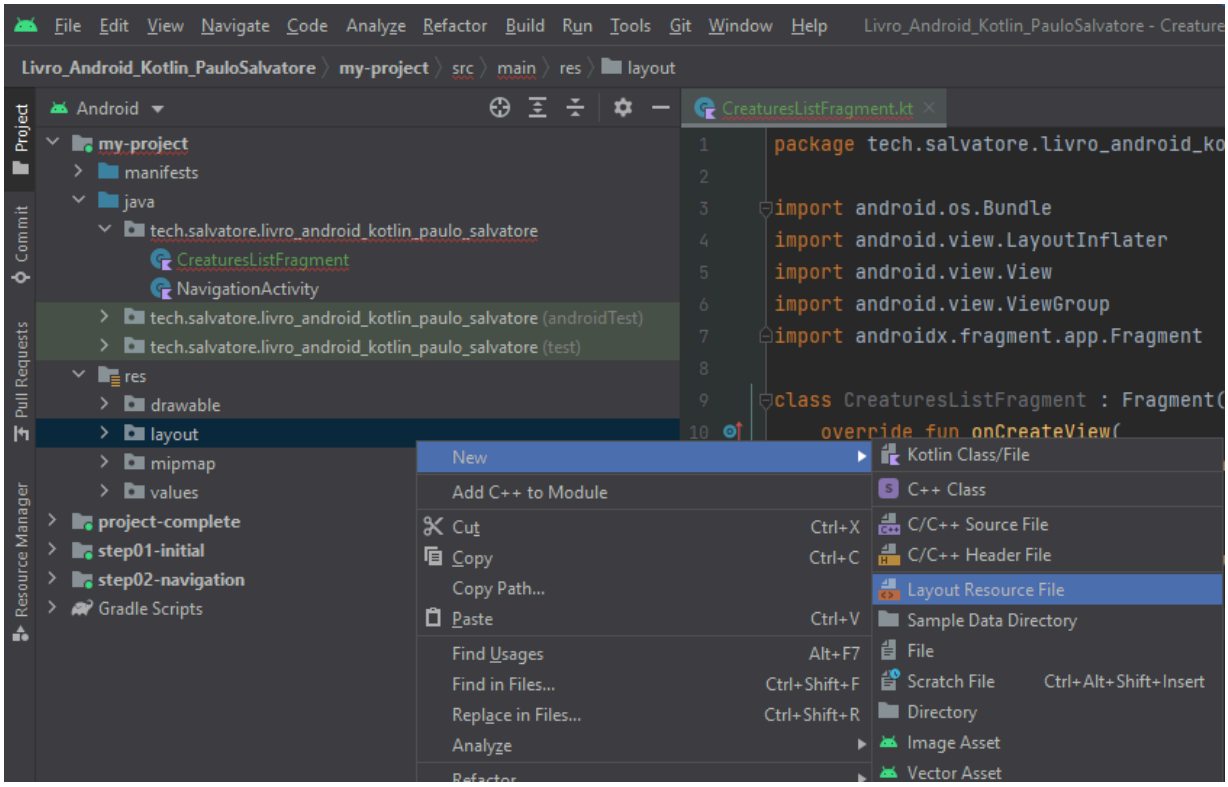


Figura 4.4: Criação do arquivo de layout do Fragment.

Nomeie o arquivo como `creatures_list_fragment.xml` e clique no botão **OK**. Com o arquivo criado, mude a visualização para o modo **Code**, em que é possível editar o conteúdo XML do arquivo. Adicione o seguinte *boilerplate*:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".CreaturesListFragment">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="CreaturesListFragment"
```

```
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:ignore="HardcodedText" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Caso apareça a mensagem de erro do box a seguir, dizendo que o layout `creatures_list_fragment` não possui uma declaração na pasta de layouts, pode ser que o Android Studio tenha se perdido com os arquivos de cache.

> *The layout "creatures_list_fragment" in layout has no declaration in the base layout folder; this can lead to crashes when the resource is queried in a configuration that does not match this qualifier.*

Para resolver esse erro, navegue até o menu `File | Invalidate Caches...` para forçar a reinicialização do Android Studio, como mostrado na imagem a seguir.

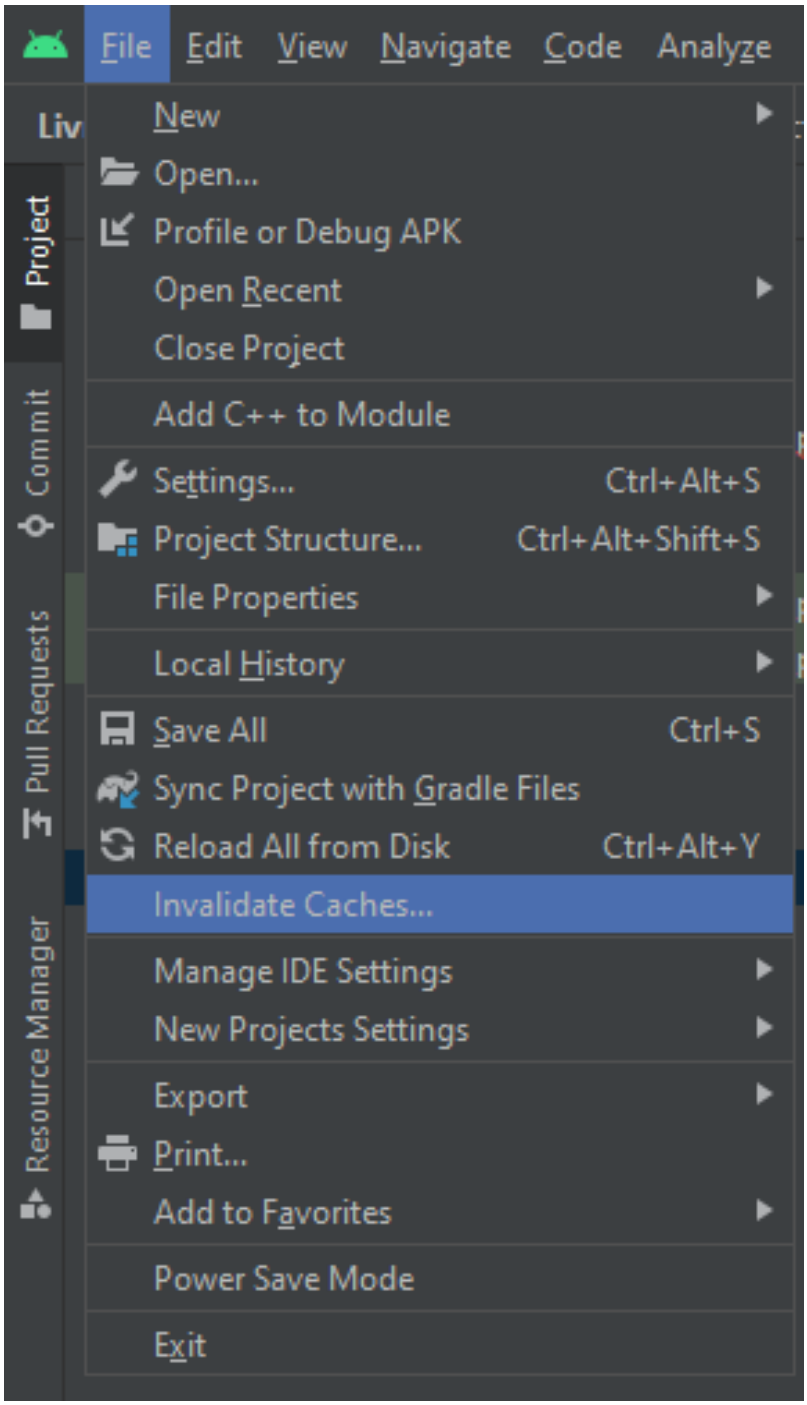


Figura 4.5: File | Invalidate Caches...

Agora que o `Fragment` de listagem das criaturas está criado, vamos criar o gráfico de navegação da aplicação.

4.2 Criando o gráfico de navegação

O gráfico de navegação é uma funcionalidade disponível através da biblioteca `Navigations` do Android Jetpack. A primeira ação para utilizar essa biblioteca no projeto é adicionar as dependências no `build.gradle`, como fizemos no capítulo anterior. Caso queira implementá-la em outros projetos, certifique-se de que as linhas a seguir estão devidamente adicionadas.

`build.gradle (app)`

```
plugins {
    // ...
    // Navigation
    id "androidx.navigation.safeargs.kotlin"
}
// ...
dependencies {
    // ...
    // Navigation
    implementation("androidx.navigation:navigation-fragment-
ktx:$navigationVersion")
    implementation("androidx.navigation:navigation-ui-
ktx:$navigationVersion")
}
```

O próximo passo é criar uma pasta chamada `navigation` dentro da pasta `res`. Para isso, clique com o botão direito na pasta `res` e selecione a opção `New | Android Resource Directory`, como mostrado na imagem a seguir.

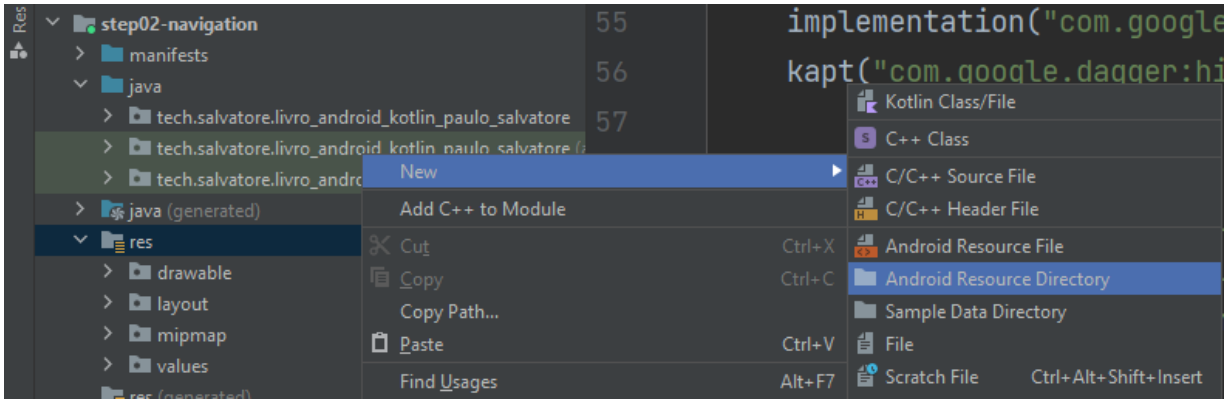


Figura 4.6: Criação do Android Resource Directory.

No menu *Resource type*, selecione a opção `navigation`, certifique-se de que o *Directory name* está com o mesmo nome e clique em **OK**.

Agora, precisamos criar o arquivo que terá o gráfico de navegação. Para isso, clique com o botão direito na pasta `navigation` e selecione a opção `New | Navigation Resource File`. Nomeie o arquivo como `nav_graph.xml` e clique em **OK**.

Criando um novo destino

Com o arquivo criado, vamos adicionar o primeiro destino da aplicação, que será a lista de criaturas. Para isso, clique no botão de `New destination` e selecione o Fragment `creatures_list_fragment`, como na imagem a seguir.

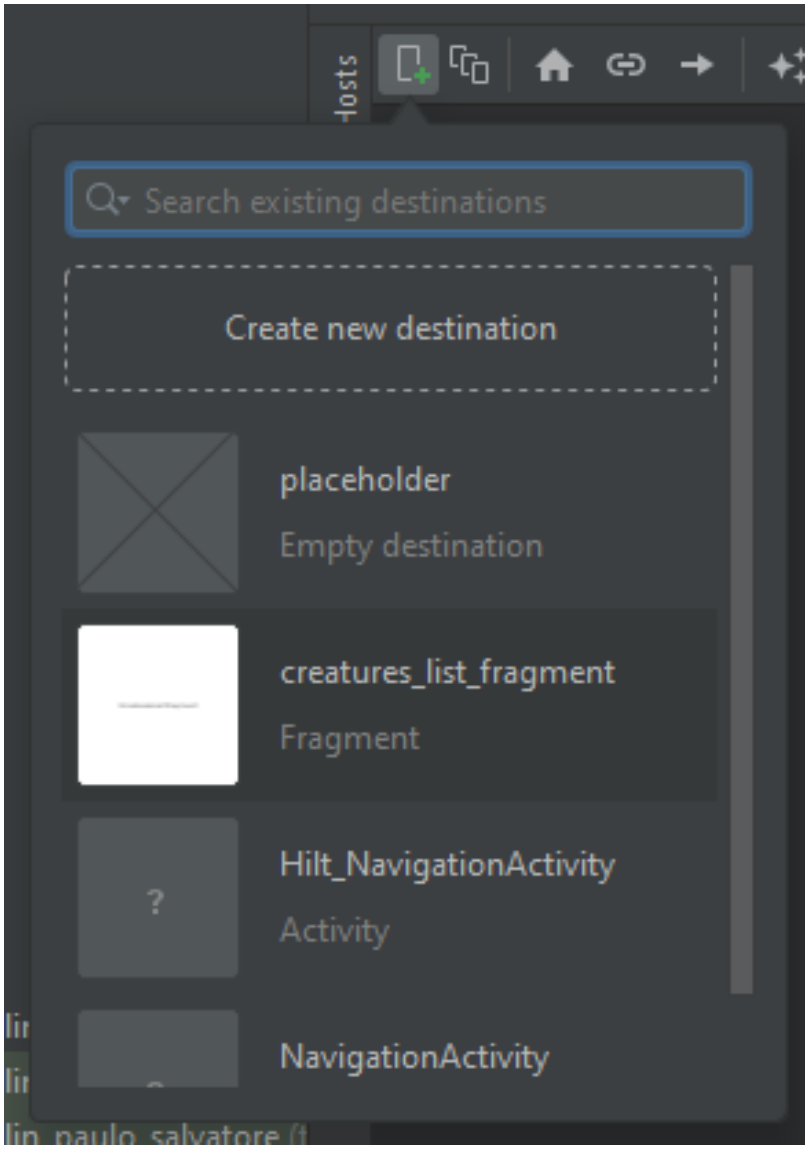


Figura 4.7: Criando um novo destino da aplicação.

Para visualizar o código gerado, clique no botão `Code` , na parte superior direita do Android Studio. O código deverá estar da seguinte maneira.

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph.xml"
    app:startDestination="@id/creaturesListFragment">
```

```
    <fragment
        android:id="@+id/creaturesListFragment"

android:name="tech.salvatore.livro_android_kotlin_paulo_salvatore.CreaturesListFragment"
        android:label="creatures_list_fragment"
        tools:layout="@layout/creatures_list_fragment" />
</navigation>
```

Caso queira manter o padrão de nomenclatura, renomeie o `id` do `Fragment` de `@+id/creaturesListFragment` para `@+id/creatures_list_dest`.

> **Dica:** para renomear a declaração e todas as utilizações, clique em cima do nome e utilize o atalho `Shift + F6`.

Agora que temos o gráfico de navegação criado e com o primeiro destino, vamos integrá-lo na `NavigationActivity` para que a lista de criaturas apareça logo no início do app.

Integrando o gráfico na Activity

Para conectar o gráfico com a `Activity`, precisamos de um `Fragment` que referencie a classe `NavHostFragment` fornecida pela biblioteca `Navigation`. A partir do `NavHostFragment`, inflamos o gráfico em XML que foi criado anteriormente.

Para declarar o `NavHostFragment`, abra o arquivo de *layout* da `NavigationActivity`, chamado `navigation_activity.xml`, e adicione o seguinte conteúdo:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
```

```
    tools:context=".NavigationActivity">

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/nav_host_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        app:defaultNavHost="true" />
</LinearLayout>
```

O próximo passo é inflar o gráfico XML a partir da `NavigationActivity`. Para isso, acessamos o `NavHostFragment` declarado, o `NavController` dentro do `Fragment`. Com acesso ao `NavController`, é possível usar o `NavInflater` para inflar o XML, informando o caminho do `nav_graph.xml` através da declaração `R.navigation.nav_graph`. Com o *graph* inflado, é hora de defini-lo como o gráfico do `NavController`, da seguinte forma:

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore

import android.os.Bundle
import androidx.activity.viewModels
import androidx.appcompat.app.AppCompatActivity
import androidx.navigation.fragment.NavHostFragment
import dagger.hilt.android.AndroidEntryPoint

@AndroidEntryPoint
class NavigationActivity : AppCompatActivity() {
    private val navigationViewModel: NavigationViewModel by viewModels()

    private val navController by lazy {
        val navHostFragment =
            supportFragmentManager
                .findFragmentById(R.id.nav_host_fragment) as
NavHostFragment

        navHostFragment.navController
    }
```

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.navigation_activity)

        initNavigation()
    }

    private fun initNavigation() {
        // Add NavGraph
        val graphInflater = navController.navInflater
        val navGraph = graphInflater.inflate(R.navigation.nav_graph)
        navController.graph = navGraph
    }
}
```

Note que as propriedades `navHostFragment` e `navController` possuem a declaração `by lazy {}` para serem inicializadas.

Propriedades que utilizam `Lazy` serão inicializadas quando forem utilizadas pela primeira vez. Isso é muito útil quando temos uma propriedade que precisa esperar a `Activity` ou o `Fragment` estarem 100% criados para poder realizar operações como o `findFragmentById`.

Com isso, rode a aplicação (`Shift + F10`) para visualizar o resultado.

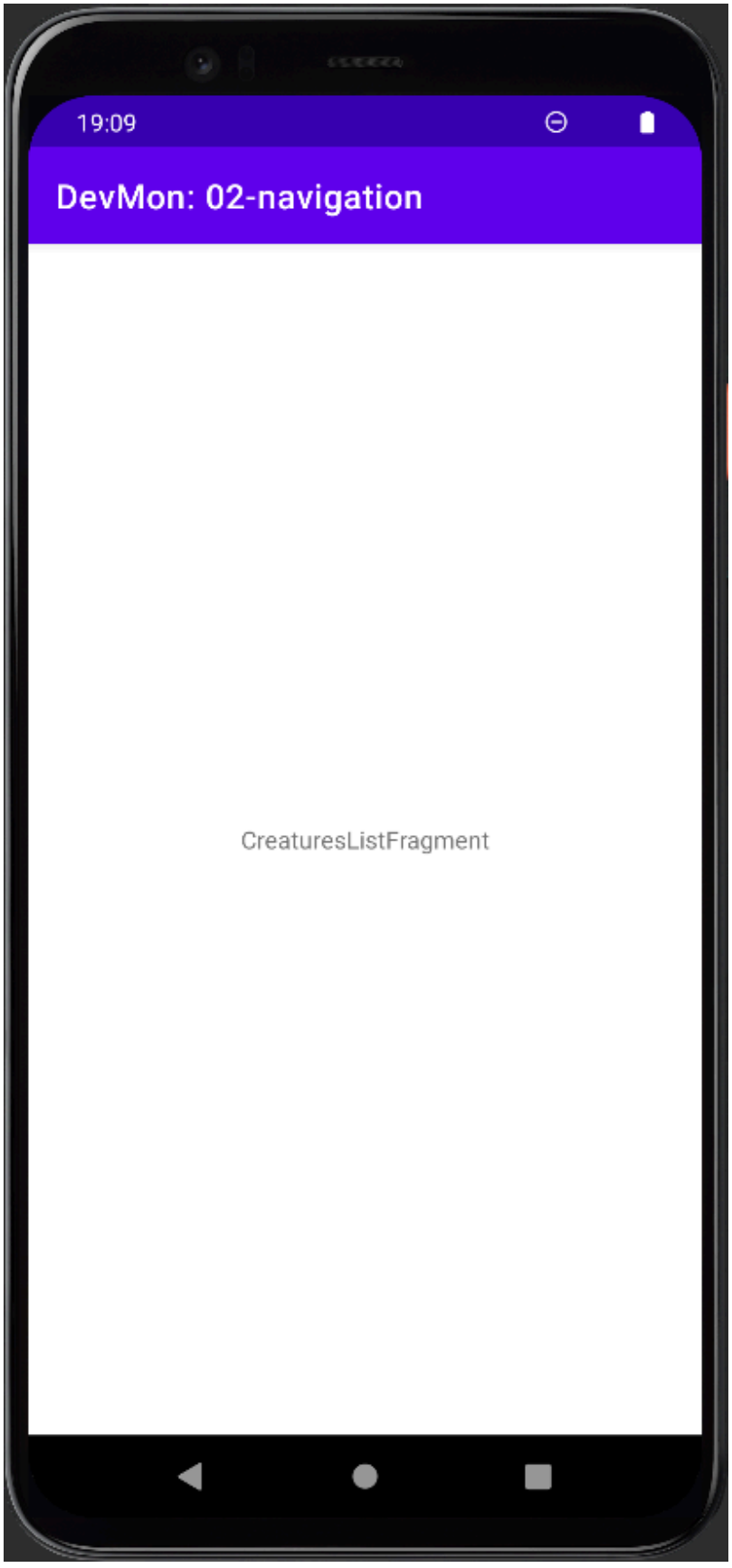

19:09
DevMon: 02-navigation
CreaturesListFragment

Figura 4.8: Fragment declarado no startDestination.

Conclusão

Temos o gráfico de navegação criado e o primeiro destino da aplicação devidamente configurado. O próximo passo é implementar a lista de criaturas e começar a organizar a arquitetura do projeto.

Por último, mas não menos importante, o DevMon #5 é o **C++**, em homenagem à linguagem de programação de mesmo nome que é extremamente poderosa e muito utilizada atualmente quando o assunto é performance. Apesar de o nome fazer referência à linguagem C, o C++ não foi escrito pela mesma equipe/empresa. O nome original, *C with Classes* (C com Classes), não foi muito popular e rapidamente foi substituído por C++, pegando carona na popularidade da linguagem C, que já era amplamente utilizada.

URL da imagem: https://i.imgur.com/YYkSF5A.png.

Figura 4.9: O DevMon #5 se chama C++!

CAPÍTULO 5
Lista de criaturas

> Caso queira visualizar as implementações completas deste capítulo, acesse o módulo `CAP05-CREATURES-LIST`.

A lista de criaturas é a principal tela da aplicação, afinal, é por ela que visualizamos todas as criaturas, escolhemos uma para mostrar seus detalhes e sabemos quando há uma nova criatura disponível.

Para implementar a lista, utilizamos a classe `RecyclerView`, que automaticamente recicla cada um dos itens da lista, o que otimiza o consumo de memória, principalmente para listas com muitos elementos. Neste capítulo, veremos como construir as classes necessárias: `RecyclerView.Adapter` e `RecyclerView.ViewHolder`.

Além disso, neste capítulo, revelaremos uma criatura muito especial, o DevMon #6, nomeado em homenagem a uma linguagem de programação desenvolvida pela Microsoft e que vem ganhando cada vez mais espaço dentro da comunidade. Assim como o Java e o Kotlin, revelados anteriormente, é uma linguagem que faz parte da minha *stack* de tecnologia, principalmente quando estou construindo jogos e aplicações em realidade virtual utilizando o motor de games Unity.

Figura 5.1: Qual é o DevMon #6?

Antes de começar a construção da lista, precisamos organizar a arquitetura do projeto, pois estamos com todas as classes direto na raiz do pacote principal da aplicação.

Organizando a arquitetura do projeto

Como vimos nos capítulos anteriores, a arquitetura da nossa aplicação é a MVVM (*Model*, *View* e *View Model*). Para que a aplicação evolua de forma saudável, é interessante organizarmos a estrutura de pastas desde o início. Para isso, seguiremos as instruções a seguir:

- **Camada *View*:** arquivos relacionados às telas do Android no geral, como `Activity`, `Fragment`, `Adapter`, arquivos de utilidades da *View* etc. Esses arquivos acessam apenas as classes da camada *View Model*.

- **Camada *View Model***: classes que estendam a classe `ViewModel` disponível no Android Jetpack. Esses arquivos acessam apenas as classes `Repository` da camada *Model*.
- **Camada *Model***: todas as classes que manipulam os dados da aplicação, sejam dados que estão em alguma API da internet, em um banco de dados local ou salvos apenas enquanto o app estiver aberto. Esses arquivos não podem acessar arquivos de outras camadas.

Por enquanto, a aplicação tem apenas as classes `App`, `CreaturesListFragment`, `NavigationActivity` e `NavigationViewModel`. Para organizá-las, deixamos a classe `App` no pacote principal da aplicação e movemos `CreaturesListFragment` e `NavigationActivity` para um novo pacote chamado `view` e a classe `NavigationViewModel` para um novo pacote chamado `viewmodel`.

Para criar um pacote, clique com o botão direito em cima do pacote principal da aplicação e selecione a opção `New | Package`, como na figura a seguir.

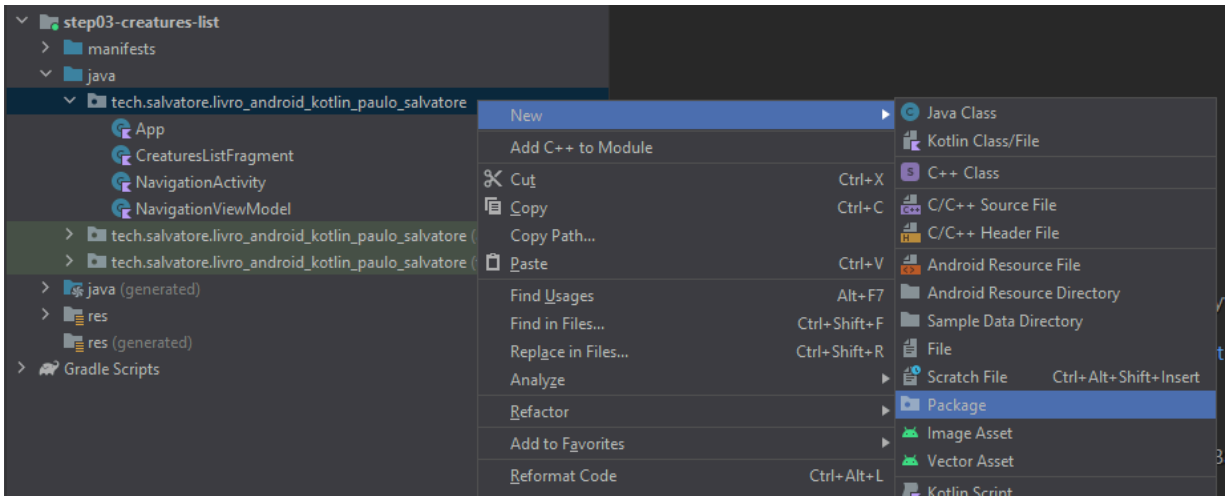


Figura 5.2: Criação de um novo package.

Crie dois pacotes: um chamado `view` e outro chamado `viewmodel`. Arraste e solte o arquivo `CreaturesListFragment` para o pacote `view`. Confirme a operação clicando no botão **Refactor**. Faça o mesmo processo para o arquivo `NavigationActivity`, movendo-o para o

mesmo pacote. Por fim, mova o arquivo `NavigationViewModel` para o pacote `viewmodel`.

> **ATENÇÃO:** em alguns casos, quando movemos arquivos, o Android Studio não atualiza corretamente a visualização. Caso isso ocorra, reiniciar o Android Studio costuma resolver o problema.

5.1 Preparando a camada Model

Agora que organizamos a estrutura de pastas do projeto, vamos aproveitar para preparar a camada *Model*, criando um pacote específico para ela e adicionando alguns arquivos que servirão para a lista de criaturas.

Clique com o botão direito no pacote principal da aplicação e crie um novo pacote chamado `model`. Dentro deste novo pacote, crie duas novas classes: `Creature` e `CreatureRepository`.

A classe `Creature` possui a estrutura das criaturas da aplicação e corresponderá a uma classe de domínio, que são extremamente importantes para qualquer software. Sempre que estamos trafegando dados da camada *Model* para as outras camadas, eles são estruturados através das classes de domínio, que de fato dominam toda a lógica da aplicação. Para começar, a classe `Creature` contém as seguintes informações: número (`number`), nome (`name`) e URL da imagem (`imageUrl`).

Já a classe `CreatureRepository` é responsável por obter e disponibilizar os dados, independente de onde eles estão, seja em uma API na internet ou em um banco de dados local. Neste capítulo, essa classe terá uma lista de criaturas armazenadas em uma variável. Nos próximos capítulos, vamos buscar esses dados em uma API na internet.

Após criar as duas classes, crie dois pacotes dentro do pacote `model`, o primeiro chamado `domain` e o segundo chamado `repository`. Mova o arquivo `Creature.kt` para o pacote `domain` e o arquivo `CreatureRepository.kt` para o pacote `repository`, como na imagem a seguir.

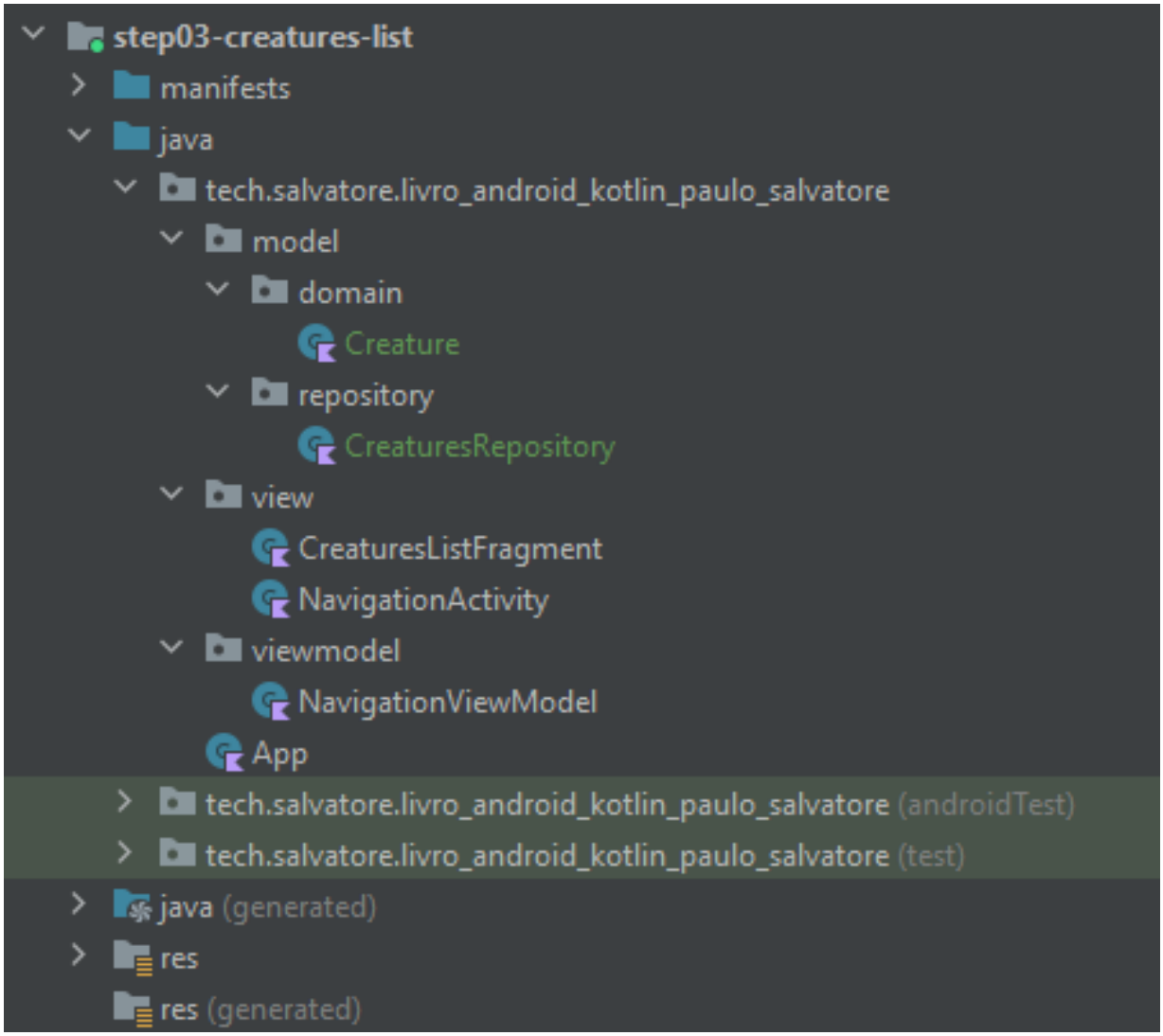


Figura 5.3: Estrutura de pastas.

Para adicionar a estrutura dos dados de uma criatura, de acordo com o que foi mencionado anteriormente, abra o arquivo `Creature.kt` recém-criado e adicione o conteúdo a seguir:

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore.model.domain

data class Creature(
    val number: Long,
    val name: String,
    val imageUrl: String,
)
```

Agora, vamos preparar a classe `CreatureRepository` para ter uma lista de criaturas que exibiremos na aplicação. Abra o arquivo `CreatureRepository.kt` e adicione o conteúdo a seguir.

```
package
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.repository

import
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.domain.Creature

class CreaturesRepository {
    val creatures: List<Creature>

    init {
        val creature1 = Creature(1, "Java",
"https://i.imgur.com/PHfd1h2.png")
        val creature2 = Creature(2, "Kotlin",
"https://i.imgur.com/EZ3zCBp.png")
        val creature3 = Creature(3, "????",
"https://i.imgur.com/rNKVxSt.png")
        val creature4 = Creature(4, "C",
"https://i.imgur.com/iWeXxA9.png")
        val creature5 = Creature(5, "C++",
"https://i.imgur.com/YYkSF5A.png")
        val creature6 = Creature(6, "????",
"https://i.imgur.com/rNKVxSt.png")

        creatures = listOf(
            creature1,
            creature2,
            creature3,
```

```
            creature4,
            creature5,
            creature6,
            // A primeira geração de DevMon possui 13 criaturas
        )
    }
}
```

Com a camada *Model* preparada, vamos entender como a comunicação desses dados deve ser feita.

Comunicando a atualização dos dados entre camadas

Como mencionado anteriormente, a camada *View Model* é responsável por acessar os dados da camada *Model* e deixá-los disponíveis para que a *View* os acesse. Além disso, dentro de uma aplicação com arquitetura MVVM, além de disponibilizar esses dados, é necessário garantir que qualquer atualização seja refletida de forma automática.

Quando estamos construindo aplicações e seguindo padrões de arquitetura, é muito importante saber as responsabilidades de cada parte do sistema. No exemplo a seguir, temos uma classe chamada `AvisosRepository`, que é responsável por armazenar as mensagens de aviso, conforme inseridas pelo usuário na `MuralAvisosActivity`.

Sempre que um aviso novo for digitado, a função `MuralAvisosActivity.novoAvisoDigitado(novoAviso: String)` será executada, o que atualizará o aviso na classe `AvisosRepository` e chamará imediatamente a função `MuralAvisosActivity.exibirAviso()`, garantindo que o `avisoTextView` esteja sempre atualizado.

> **ATENÇÃO:** o código a seguir é apenas um exemplo ilustrativo. Não é necessário criar os arquivos e implementar os códigos em seu projeto.

```
class AvisosRepository {
    var aviso: String = "Aviso inicial"

    fun atualizarAviso(novoAviso: String) {
        aviso = novoAviso
    }
}

class MuralAvisosActivity {
    private val avisosRepository = AvisosRepository()

    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
        exibirAviso()
    }

    fun exibirAviso() {
        avisoTextView.text = avisosRepository.aviso
    }

    fun novoAvisoDigitado(novoAviso: String) {
        avisosRepository.atualizarAviso(novoAviso)
        exibirAviso()
    }
}
```

Apesar de esse exemplo de código funcionar como esperado, ele possui alguns problemas:

1. Sempre que atualizarmos algum aviso, devemos manualmente executar a função `exibirAviso()` para garantir que o `textView` possua o valor mais atualizado possível.
2. Se o aviso for atualizado por alguma outra classe que não seja a classe `MuralAvisosActivity`, o valor do `textView` ficará desatualizado.

Na situação número 2, uma forma de resolver esse problema seria fazer com que a classe `AvisosRepository` avise a classe `MuralAvisosActivity` de que a mensagem de aviso foi atualizada, do

mesmo jeito que a classe `MuralAvisosActivity` está executando diretamente a função `avisosRepository.atualizarAviso(novoAviso)`. O problema, nesse caso, é que a camada `model` não deve possuir referência a nenhuma classe que não esteja na mesma camada, o que impede esse tipo de implementação.

Para resolver esse tipo de problema, um recurso muito utilizado é a criação de eventos. Dessa forma, sempre que a mensagem de aviso for atualizada, um novo evento é emitido pela classe `AvisosRepository` informando que há uma nova mensagem disponível. Com isso, a classe `MuralAvisosActivity` pode ouvir os eventos de atualização, receber a nova mensagem e atualizar no `textView`.

No entanto, esse tipo de abordagem causa um novo problema: quando a `MuralAvisosActivity` for destruída, os eventos da `AvisosRepository` continuam sendo ouvidos, causando vazamentos de memória, a menos que deixemos explícito para parar de ouvi-lo. A solução é simples, mas acaba se tornando um *boilerplate* presente em muitos lugares da aplicação.

> Vazamentos de memória (*Memory Leak*) consistem em variáveis e declarações que não foram removidas após não serem mais necessárias. Isso faz com que softwares fiquem mais lentos após determinado tempo aberto. Devs de alto nível devem se preocupar com esse tipo de situação, identificando e resolvendo possíveis vazamentos de memória no código.

Para facilitar a utilização desse tipo de abordagem, existem diversas bibliotecas que permitem a utilização de objetos "observáveis", também conhecidos como `Observables`. Neste livro, utilizaremos três bibliotecas que possuem esse mecanismo: *Data Binding*, *LiveData* e *RX*. É válido ressaltar que cada biblioteca possui uma característica específica e costuma servir para resolver o mesmo problema em camadas diferentes. Por exemplo:

- **Data Binding:** comunicação entre o *layout* XML e a classe da `Activity`;
- **LiveData:** comunicação entre a classe da `Activity` e o `ViewModel`;
- **RX:** comunicação entre a camada *Model* e a camada *View Model*.

Neste capítulo, trataremos apenas da implementação do *LiveData*, pois o utilizaremos para realizar a comunicação entre `CreaturesViewModel` (que criaremos mais à frente) e `CreaturesListFragment`.

Como vimos, o Android Jetpack possui alguns componentes dentro da categoria *Architecture*. Dentre eles, três são importantes para este capítulo: *Lifecycles*, *LiveData* e *View Model*.

O componente *Lifecycles* é composto de duas interfaces principais, `LifecycleOwner` e `LifecycleRegistryOwner`. Ambas são implementadas nas classes `AppCompatActivity` e `Fragment`. É possível atrelar outros componentes para objetos que implementam essas interfaces (que chamaremos de `owners`) para detectar mudanças no ciclo de vida do seu respectivo `owner`. Todo objeto que implementar o `LifecycleOwner`, incluindo qualquer `Activity` ou `Fragment`, possuirá o seu próprio ciclo de vida.

O *LiveData* é uma implementação bem simples que permite observar qualquer mudança de uma informação. Sempre que essa informação for alterada, o *LiveData* informará esse evento para todos os interessados. Para isso, quem tiver interesse deverá criar um `Observer`, que registra no *LiveData* qual a ação que deverá ser executada sempre que o dado mudar. É importante mencionar que o *LiveData* respeita o ciclo de vida dos componentes, incluindo `Activities`, `Fragments`, `Services` e qualquer outro `LifecycleOwner` da aplicação. Isso faz com que, quando o `LifecycleOwner` for destruído, tudo que estiver sendo observado seja removido automaticamente.

5.2 Preparando a camada View Model

Como mencionado anteriormente, a camada *View Model* é responsável por acessar os dados da camada *Model* e deixá-los disponíveis para que a *View* os acesse. Dentro de uma aplicação com arquitetura MVVM, além de disponibilizar esses dados, é necessário garantir que qualquer atualização seja refletida de forma automática. Como estamos realizando uma comunicação entre `Activity` e `ViewModel`, é interessante utilizar a classe `LiveData` para, além de permitir que atualizações sejam observadas, aproveitarmos o ciclo de vida da `Activity` para remover tudo o que estiver sendo observado quando a tela for destruída.

Para disponibilizar a lista de criaturas que está na classe `CreaturesRepository` na `CreaturesListFragment`, vamos criar uma nova classe, chamada `CreaturesViewModel`, acessar o *Repository* e disponibilizar a lista de criaturas através de uma propriedade com `LiveData`. No pacote `viewmodel`, crie um novo arquivo chamado `CreaturesViewModel.kt` e adicione o conteúdo a seguir:

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore.viewmodel

import androidx.lifecycle.ViewModel
import dagger.hilt.android.lifecycle.HiltViewModel
import javax.inject.Inject

@HiltViewModel
class CreaturesViewModel @Inject constructor() : ViewModel() {
}
```

Para acessarmos a classe `CreaturesRepository` dentro de `CreaturesViewModel`, podemos utilizar o mecanismo de injeção de dependência do Hilt. Para isso, precisamos declarar dentro do construtor as classes necessárias:

```
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.repository.Creat
uresRepository
```

```
@HiltViewModel
class CreaturesViewModel @Inject constructor(
    creaturesRepository: CreaturesRepository
) : ViewModel() {
}
```

No entanto, para que o Hilt saiba como fornecer uma instância da classe `CreaturesRepository`, precisamos adicionar a *annotation* `@Inject` em um construtor vazio. Além disso, classes de repository costumam ter apenas uma instância na aplicação toda, que é reaproveitada pelas demais classes do sistema. Para garantir que o Hilt vai fornecer a mesma instância sempre, adicionaremos também a *annotation* `@Singleton` na classe.

```
import javax.inject.Inject
import javax.inject.Singleton

@Singleton
class CreaturesRepository @Inject constructor() {
//...
```

Agora que temos acesso à instância de `CreaturesRepository` na classe `CreaturesViewModel`, vamos preparar a propriedade `LiveData` para ser acessada pela classe `CreaturesListFragment`. A classe `LiveData` é uma classe abstrata (`abstract`), ou seja, ela não pode ser utilizada diretamente, a menos que tenha uma implementação concreta. Felizmente, há uma classe chamada `MutableLiveData` que traz uma implementação concreta da classe `LiveData`. Para utilizá-la, basta adicioná-la como tipo de uma propriedade, informando qual o tipo do dado que será armazenado.

Como estamos preparando uma lista de criaturas, vamos passar o tipo `List<Creature>` para a propriedade `MutableLiveData`. Além disso, quando o `ViewModel` for inicializado, pegaremos a lista de criaturas na classe `CreaturesRepository` e atualizaremos o valor da propriedade `LiveData`, da seguinte forma:

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore.viewmodel

import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import dagger.hilt.android.lifecycle.HiltViewModel
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.domain.Creature
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.repository.Creat
uresRepository
import javax.inject.Inject

@HiltViewModel
class CreaturesViewModel @Inject constructor(
    creaturesRepository: CreaturesRepository
) : ViewModel() {
    val creatures = MutableLiveData<List<Creature>>()

    init {
        creatures.value = creaturesRepository.creatures
    }
}
```

> Note que, para mudar o valor da propriedade `creatures`, utilizamos a declaração `creatures.value =`, que, por baixo dos panos, utiliza o método `setValue()` da classe `MutableLiveData`, escrita na linguagem Java. É válido mencionar que, se estivéssemos em uma *thread* separada, seria necessário utilizar a declaração `postValue()` do `MutableLiveData`. Nos próximos capítulos, trataremos um pouco mais do assunto de *threads*.

Agora que os dados estão preparados na camada *View Model*, podemos acessá-los na camada *View*.

5.3 Integrando Model e View Model na camada View

O primeiro passo para acessar os dados na camada *View* é inicializar a classe `CreaturesViewModel` utilizando a declaração `by viewModels()`, como fizemos anteriormente. Além disso, para que o Hilt consiga fazer a injeção no `ViewModel`, precisamos anotar o `Fragment` com `@AndroidEntryPoint`.

Abra o arquivo `CreaturesListFragment` e adicione a seguinte declaração:

```
import androidx.fragment.app.viewModels
import dagger.hilt.android.AndroidEntryPoint

@AndroidEntryPoint
class CreaturesListFragment : Fragment() {
    private val creaturesViewModel: CreaturesViewModel by viewModels()
    // ...
```

Para acessar a lista de criaturas sempre que ela for atualizada, basta declarar `creaturesViewModel.creatures` em qualquer lugar do `Fragment` e chamar o método `observe` passando duas informações. A primeira é o `LifecycleOwner` (pode ser uma `Activity`, `Fragment` etc.); o `LiveData` para de enviar informações quando o `LifecycleOwner` for destruído. A segunda é uma implementação do `Observer`, que recebe o novo valor que foi alterado e nos permite inserir uma lógica de apresentação que será executada sempre que o `LiveData` mudar.

Quando estamos dentro de um `Fragment`, um dos melhores lugares para esse tipo de declaração é no método do ciclo de vida chamado `onViewCreated()`, que é executado assim que o método `onCreateView()` finaliza a criação do *layout*, da seguinte forma:

```
import androidx.lifecycle.Observer

@AndroidEntryPoint
class CreaturesListFragment : Fragment() {
```

```
    private val creaturesViewModel: CreaturesViewModel by viewModels()
    // ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        creaturesViewModel.creatures.observe(viewLifecycleOwner, Observer
{
            // it = List<Creature>
        })
    }
}
```

No Kotlin, podemos simplificar a declaração removendo a palavra `Observer`, visto que é possível determinar o tipo por associação. Ao remover, transformamos a declaração em um argumento `lambda`, que pode ser removido dos parênteses, ficando assim:

```
// ...
class CreaturesListFragment : Fragment() {
    private val creaturesViewModel: CreaturesViewModel by viewModels()
    // ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        creaturesViewModel.creatures.observe(viewLifecycleOwner) {
            // it = List<Creature>
        }
    }
}
```

Ao utilizar o `lambda` como argumento no lugar da classe `Observer`, a linha de `import` pode ser removida. É possível utilizar o atalho `Ctrl + Alt + O` para remover `imports` que não estão sendo utilizados.

> Note que, em ambos os casos, dentro das chaves `{ }`, temos o operador `it` contendo a lista de criaturas. O `it` é um operador auxiliar do Kotlin que aparece em declarações `lambda` com apenas um parâmetro, como é o caso do `Observer`, que traz apenas um parâmetro com a informação que acabou de ser atualizada no `LiveData`.

Agora que temos acesso à lista de criaturas, podemos criar um componente visual para exibir todas as criaturas disponíveis. Como mencionamos anteriormente, o componente utilizado para a exibição de listas no Android é o `RecyclerView`.

Implementando o `RecyclerView`

Praticamente todas as aplicações de Android existentes hoje possuem uma lista em algum momento da aplicação. No início do Android, o elemento utilizado para tal função era o `ListView`, porém, ele possui uma série de limitações em termos de otimização e reaproveitamento de recursos. A boa prática para a construção de listas é utilizar o elemento `RecyclerView`, que faz uma boa gestão dessas informações.

Para implementar a `RecyclerView`, precisamos realizar os seguintes passos:

- Criar um arquivo de *layout* XML para cada item da lista;
- Criar um arquivo para o `RecyclerView.Adapter` contendo uma implementação de `RecyclerView.ViewHolder`;
- Declarar o componente `RecyclerView` no arquivo de *layout* do `CreaturesListFragment`.

Vamos começar criando o arquivo de *layout* de cada item da lista. Ele ficará como a imagem a seguir.

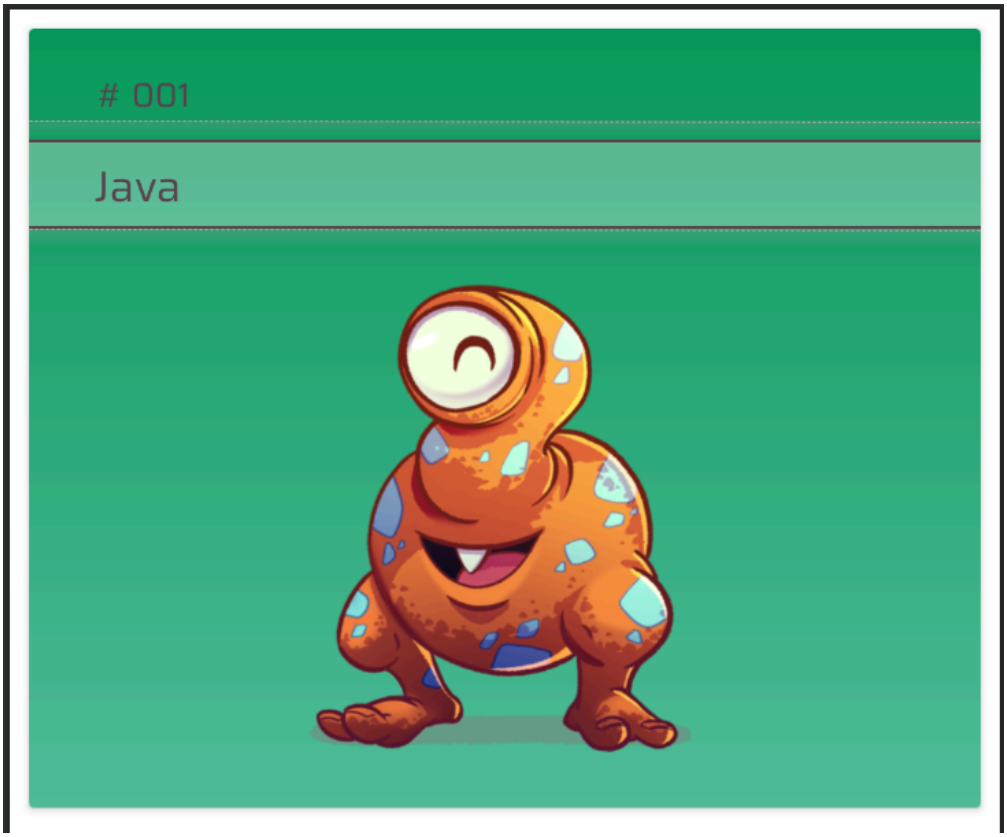


Figura 5.4: Preview do layout dos itens da lista.

A estrutura desse *layout* segue este fluxo:

1. Elemento da raiz é um `CardView`, que traz os cantos arredondados e um pouco de *shadow* ao elemento;
2. Logo no primeiro elemento, temos um `ConstraintLayout`, que facilita a organização e a estruturação dos outros elementos. O *background* deste elemento será o recurso `screen_background.xml`, que já está importado na pasta `res/drawable` do módulo `my-project`.
3. O número da criatura é um `TextView` simples. Todos os textos da aplicação utilizarão a fonte Saira, que já está importada no módulo `my-project`, dentro da pasta `res/font`.
4. Abaixo do número, utilizamos o elemento `Barrier`, do `ConstraintLayout`, que constrói uma barreira invisível que auxilia na separação de *layouts* e elementos mais complexos. Dessa

forma, as partes inferiores do *layout* se conectam a essa barreira, em vez de dependerem de outros elementos da tela.

5. Para construir o nome da criatura, utilizamos um `ConstraintLayout` contendo um `TextView`. O `ConstraintLayout` utiliza um estilo que já está presente na aplicação, chamado `creatures_background_title_section`, que é utilizado em todos os lugares que exibem o nome da criatura. Esse estilo, que está no arquivo `res/values/styles.xml`, utiliza um recurso de *background* chamado `background_creature_title_section`, que está importado na pasta `res/drawable`.
6. Uma nova `Barrier` é adicionada seguindo o mesmo conceito da primeira;
7. Por fim, temos um `ImageView` que exibe a imagem da criatura.

> Fique à vontade para construir o *layout* seguindo o fluxo proposto. Neste livro, não abordaremos a construção dos *layouts* passo a passo, apenas quando tiver alguma relação com uma implementação mais específica.

Para criar o *layout*, clique com o botão direito na pasta `layout`, que fica dentro da pasta `res`. Em seguida, clique na opção `New | Layout Resource File`. Nomeie o arquivo com `creatures_list_item.xml`, mantenha todas as outras opções como estão e clique em **OK**.

Dentro do novo arquivo criado, mude para a visão do código, clicando no botão `Code` na parte superior direita da tela. Adicione o código a seguir:

Código disponível no GitHub através do link:

https://github.com/paulosalvatore/Livro_Android_Kotlin_PauloSalvatore/blob/develop/step03-creatures-list/src/main/res/layout/creatures_list_item.xml

Figura 5.5: Código do Layout do Item da Lista.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    android:padding="8dp">

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="@drawable/screen_background">
```

```
<TextView
    android:id="@+id/tvCreatureNumber"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="28dp"
    android:layout_marginLeft="28dp"
    android:layout_marginTop="16dp"
    android:fontFamily="@font/saira_400"
    android:text="# 001"
    android:textColor="#5D414F"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<androidx.constraintlayout.widget.Barrier
    android:id="@+id/barrier1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:barrierDirection="bottom"
    app:constraint_referenced_ids="tvCreatureNumber" />

<androidx.constraintlayout.widget.ConstraintLayout
    android:id="@+id/constraintLayoutTitleSection"
    style="@style/creatures_background_title_section"
    android:layout_height="37dp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/barrier1">

    <TextView
        android:id="@+id/tvCreatureName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="27dp"
        android:layout_marginLeft="27dp"
        android:fontFamily="@font/saira_400"
        android:text="Java"
        android:textColor="#5D414F"
        android:textSize="18sp"
        app:layout_constraintBottom_toBottomOf="parent"
```

```
                app:layout_constraintStart_toStartOf="parent"
                app:layout_constraintTop_toTopOf="parent" />

        </androidx.constraintlayout.widget.ConstraintLayout>

        <androidx.constraintlayout.widget.Barrier
            android:id="@+id/barrier3"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            app:barrierDirection="bottom"
            app:constraint_referenced_ids="constraintLayoutTitleSection"
/>

        <ImageView
            android:id="@+id/ivCreature"
            android:layout_width="200dp"
            android:layout_height="200dp"
            android:layout_marginTop="20dp"
            android:layout_marginBottom="20dp"
            android:contentDescription="TODO: Content Description"
            android:src="@drawable/creature_java"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@id/barrier3" />

    </androidx.constraintlayout.widget.ConstraintLayout>
</androidx.cardview.widget.CardView>
```

Com o *layout* pronto, vamos criar um arquivo para o `RecyclerView.Adapter` contendo uma implementação de `RecyclerView.ViewHolder`. Vá até o pacote principal da aplicação e, dentro do pacote `view`, crie uma nova classe Kotlin chamada `CreaturesListAdapter`.

O primeiro passo é fazer com que a classe que acabamos de criar estenda a classe `RecyclerView.Adapter`:

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore.view

import androidx.recyclerview.widget.RecyclerView

class CreaturesListAdapter : RecyclerView.Adapter<>() {
}
```

A `RecyclerView` do Android depende de um recurso chamado `ViewHolder`, que, como o próprio nome diz, guarda uma `View` que, nesse caso, é o *layout* de cada item da lista. Portanto, precisamos criar uma classe que fará o papel de `ViewHolder`:

```
//...
class CreaturesListAdapter : RecyclerView.Adapter<>() {

    inner class ViewHolder {
    }
}
```

Para que essa nova classe seja reconhecida como um `ViewHolder`, precisamos estendê-la com a classe `RecyclerView.ViewHolder`. Após isso, adicionamos uma referência a essa classe dentro dos `<>` do `RecyclerView.Adapter`, como a seguir:

```
//...
class CreaturesListAdapter :
RecyclerView.Adapter<CreaturesListAdapter.ViewHolder>() {

    inner class ViewHolder : RecyclerView.ViewHolder() {
    }
}
```

No entanto, para que o `ViewHolder` funcione, é necessário passar o `itemView` que será guardado. Fazemos isso da seguinte forma:

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore.view

import android.view.View
import androidx.recyclerview.widget.RecyclerView
```

```
class CreaturesListAdapter :
RecyclerView.Adapter<CreaturesListAdapter.ViewHolder>() {

    inner class ViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
    }
}
```

Note que há o seguinte erro na classe `CreaturesListAdapter`:

> *Class 'CreaturesListAdapter' is not abstract and does not implement abstract base class member public abstract fun onCreateViewHolder(parent: ViewGroup, viewType: Int): CreaturesListAdapter.ViewHolder defined in androidx.recyclerview.widget.RecyclerView.Adapter*

Esse erro ocorre porque a classe `RecyclerView.Adapter` é abstrata e possui métodos que precisam ser implementados para que seja possível utilizá-la. Há outra opção de resolver isso, que é transformar a classe `CreaturesListAdapter` em uma classe abstrata. No entanto, isso não faz sentido nesse caso, pois, para que ela possa ser utilizada, é necessária uma implementação concreta, que é o oposto de implementações abstratas.

Para resolver esse erro, clique no nome da classe, pressione o atalho `Alt + Enter` e selecione a opção `Implement members`. Uma caixa de diálogo deve aparecer contendo três elementos. Selecione os três elementos, como na imagem a seguir, e clique no botão **OK**.

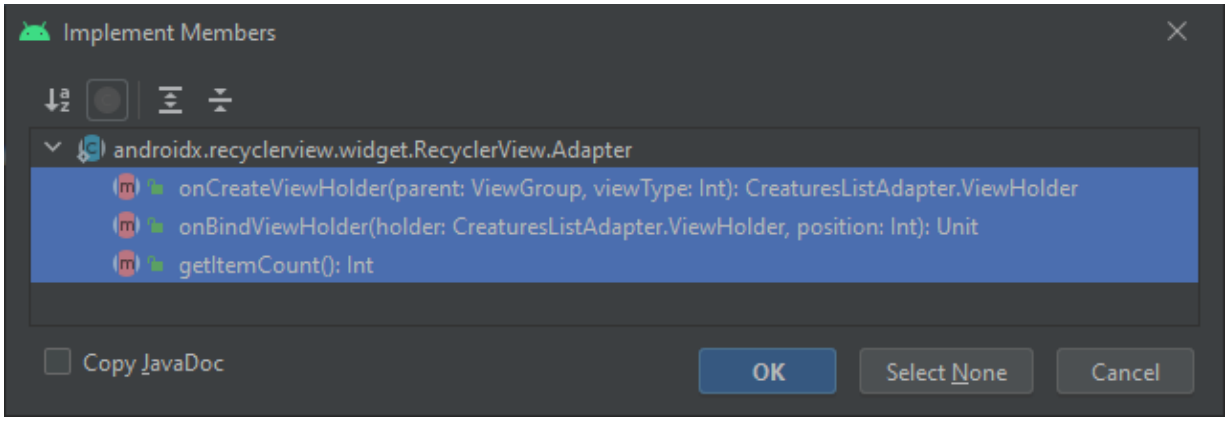


Figura 5.6: RecyclerView.Adapter, Implement Members.

Vamos começar implementando o `onCreateViewHolder` que, como o nome diz, cria as instâncias da classe `ViewHolder`, construindo as `Views` a partir do *layout* XML. Assim como as `Activities` e os `Fragments`, para criar uma `View` a partir de um arquivo XML, precisamos inflar esse arquivo utilizando a sua referência na classe `R` (que no caso é `R.layout.creatures_list_item`), o local em que essa `View` ficará disponível (representado pelo parâmetro `parent: ViewGroup`) e definindo a opção `attachToRoot` como `false`. Vamos ao código:

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore.view

// ...
import android.view.LayoutInflater
import tech.salvatore.livro_android_kotlin_paulo_salvatore.R

class CreaturesListAdapter :
RecyclerView.Adapter<CreaturesListAdapter.ViewHolder>() {
    // ...
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ViewHolder {
        val view = LayoutInflater.from(parent.context)
            .inflate(R.layout.creatures_list_item, parent, false)

        return ViewHolder(view)
```

```
    }
    // ...
```

No entanto, essas `Views` que foram construídas não possuem diferenças entre si. Para diferenciar cada item de uma lista, precisamos informar quais são os dados que devem ser atualizados. Para isso, precisamos de três elementos:

- No construtor do `CreaturesListAdapter`, devemos receber um parâmetro do tipo `List<Creature>` que possui todos os itens a serem exibidos;
- Implementar a função `getItemCount()`, que retorna quantos itens a lista possui;
- A função que atualiza os dados, o `onBindViewHolder()`, recebe a instância do `ViewHolder` que foi criada e a posição do item na lista. Com essas informações, obtemos o item específico da lista e executamos uma função dentro do `ViewHolder` para atualizar os elementos do *layout*.

> Neste capítulo, a imagem da criatura não será alterada. Trataremos da atualização das imagens no próximo capítulo.

Vamos começar adicionando a lista de itens no construtor do `CreaturesListAdapter`:

```
// ...
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.domain.Creature

class CreaturesListAdapter(
    private val items: List<Creature>,
) : RecyclerView.Adapter<CreaturesListAdapter.ViewHolder>() {
    // ...
```

Agora, alteramos o `getItemCount()` para retornar a quantidade de itens da lista:

```
// ...
class CreaturesListAdapter(
    // ...
    override fun getItemCount(): Int {
        return items.count()
    }
}
```

Com o Kotlin, utilizando o operador `=`, podemos simplificar funções que possuem apenas uma única linha que retorna algum valor, da seguinte forma:

```
override fun getItemCount() = items.count()
```

Por fim, implementamos o `onBindViewHolder()` para obter o item da lista e executar uma nova função do `ViewHolder`, que chamaremos de `bindView()`, recebendo o item e atualizando os elementos do *layout*, desta forma:

```
// ...
import android.widget.TextView

class CreaturesListAdapter(
    // ...
    inner class ViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
        fun bindView(item: Creature) {
            val tvCreatureNumber = itemView.findViewById<TextView>
(R.id.tvCreatureNumber)
            val tvCreatureName = itemView.findViewById<TextView>
(R.id.tvCreatureName)

            tvCreatureNumber.text = item.number.toString()
            tvCreatureName.text = item.name
        }
    }
    // ...
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val item = items[position]
```

```
        holder.bindView(item)
    }
```

Pronto! Com essas implementações, temos a primeira versão do `CreaturesListAdapter` implementada. Para visualizar o resultado, faltam duas etapas:

- Adicionar o elemento `RecyclerView` no arquivo de *layout* `creatures_list_fragment.xml`;
- Criar uma instância de `CreaturesListAdapter` dentro da classe `CreaturesListFragment`, passando a lista de criaturas que está disponível na propriedade `LiveData` do `CreaturesViewModel`.

Primeiro, vamos implementar o elemento `RecyclerView`. Abra o arquivo `creatures_list_fragment.xml` e adicione o seguinte conteúdo:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".view.CreaturesListFragment">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recyclerView"
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:listitem="@layout/creatures_list_item" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Caso o Android Studio esteja acusando algum erro na declaração do `android:id`, você pode tentar reiniciá-lo ou utilizar a opção do menu `Invalidate Caches...`, como mostrado anteriormente.

Depois, vamos implementar a criação do `CreaturesListAdapter` dentro da função `onViewCreated` da classe `CreaturesListFragment`:

```
// ...
class CreaturesListFragment : Fragment() {
    // ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        val recyclerView = view.findViewById<RecyclerView>
(R.id.recyclerView)

        recyclerView.layoutManager =
            LinearLayoutManager(context, LinearLayoutManager.VERTICAL,
false)

        creaturesViewModel.creatures.observe(viewLifecycleOwner) {
            recyclerView.adapter = CreaturesListAdapter(it)
        }
    }
}
```

Pronto! Agora que tudo está implementado, rode a aplicação e veja o resultado, ilustrado na imagem a seguir.

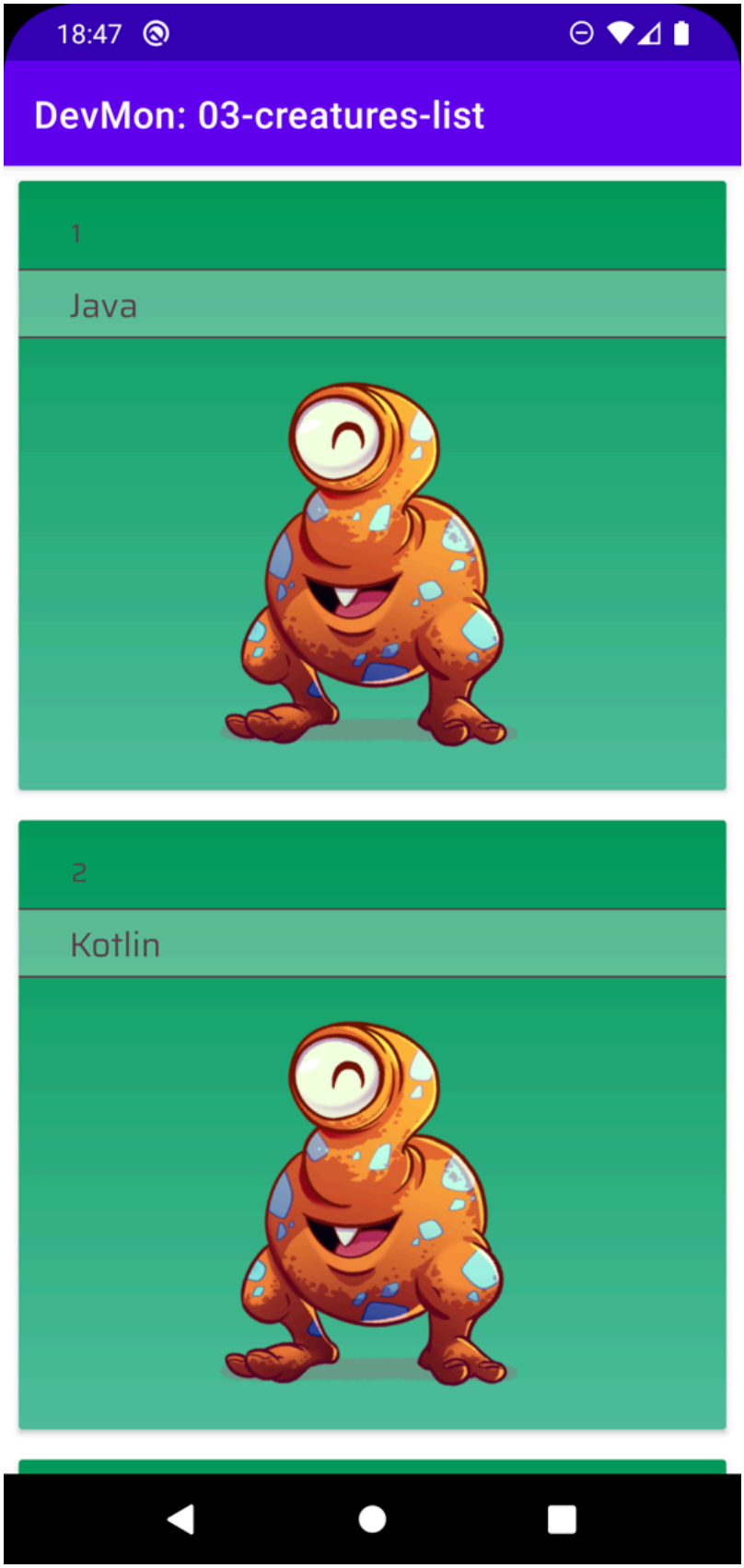

18:47
DevMon: 03-creatures-list
1
Java
2
Kotlin

Figura 5.7: Resultado final.

Conclusão

Com todos os conceitos apresentados neste capítulo, foi possível entender que cada classe possui uma responsabilidade específica, além de ter sido possível visualizar o fluxo de comunicação entre as diferentes camadas da aplicação.

Por fim, implementamos uma `RecyclerView`, que reaproveita os *layouts* e carrega as criaturas de acordo com a lista inicial que é fornecida. No quesito *layout*, ainda há espaço para melhorias: precisamos alterar as imagens de acordo com a URL de cada criatura, formatar um pouco melhor o número da criatura (`# 001`), seguindo a referência do *layout*, e permitir que, ao clicar em cada item, uma nova tela seja aberta mostrando mais detalhes da criatura.

No próximo capítulo, veremos como implementar o carregamento das imagens de cada criatura, além de ativar o recurso de *Data Binding*, que é uma das recomendações quando estamos utilizando o padrão de arquitetura MVVM.

Agora chegou a hora de revelar que o nome do DevMon deste capítulo é em homenagem à linguagem de programação **C#**, uma linguagem que está cada vez mais popular, principalmente pela sua utilização dentro do .NET Framework, com aplicações em back-end, softwares desktop, aplicativos mobile com Xamarin, jogos com Unity, entre diversas outras.

URL da imagem: https://i.imgur.com/Jldgrr6.png

Figura 5.8: O DevMon #6 se chama C#!

CAPÍTULO 6
Data Binding

> Caso queira visualizar as implementações completas deste capítulo, acesse o módulo `CAP06-DATA-BINDING`.

Como mencionado anteriormente, o Data Binding é dos recursos fundamentais quando estamos utilizando arquitetura MVVM no projeto. Ele permite abstrair boa parte da lógica de apresentação em declarações que são inseridas diretamente no arquivo de *layout* XML, evitando a escrita de muito *boilerplate*, como o uso do `findViewById` ou até de casos um pouco mais complexos do que apenas a atualização de texto de um elemento da *View*.

Antes de começar a implementação, vamos falar sobre a criatura que será revelada neste capítulo, o DevMon #7, nomeado em homenagem a uma linguagem de programação amplamente utilizada, principalmente no desenvolvimento web front-end e back-end e, também, na construção de aplicações desktop e mobile. É uma linguagem que, assim como o C#, faz parte da minha stack e provavelmente também deve fazer parte da sua.

Figura 6.1: Qual é o DevMon #7?

6.1 Implementando o Data Binding

Para implementar esse conceito no Android, temos a biblioteca Data Binding dentro dos componentes de *Architecture* do Android Jetpack. Para ativar o recurso no projeto, abra o arquivo `build.gradle` do módulo e, no final do bloco `android {`, adicione o seguinte trecho:

```
android {
    // ...
    buildFeatures {
        dataBinding = true
    }
}
```

Com as linhas adicionadas, clique em `Sync Now`.

Com o `dataBinding` habilitado, precisamos configurar o *layout* da aplicação. Por padrão, nenhum arquivo de *layout* está preparado para o *binding*. Isso acontece pois nem todas as telas possuem informações que serão atualizadas dinamicamente.

O Android Studio facilita o processo de ativar o *binding* para um *layout*: basta abrir o arquivo desejado, clicar em alguma `tag` na raiz do XML, apertar `Alt + Enter` e selecionar a opção `Convert to data binding layout`.

Para realizar o primeiro teste com o *Data Binding*, abra o arquivo `creatures_list_item.xml` e realize o processo mencionado anteriormente, como mostra a imagem a seguir.

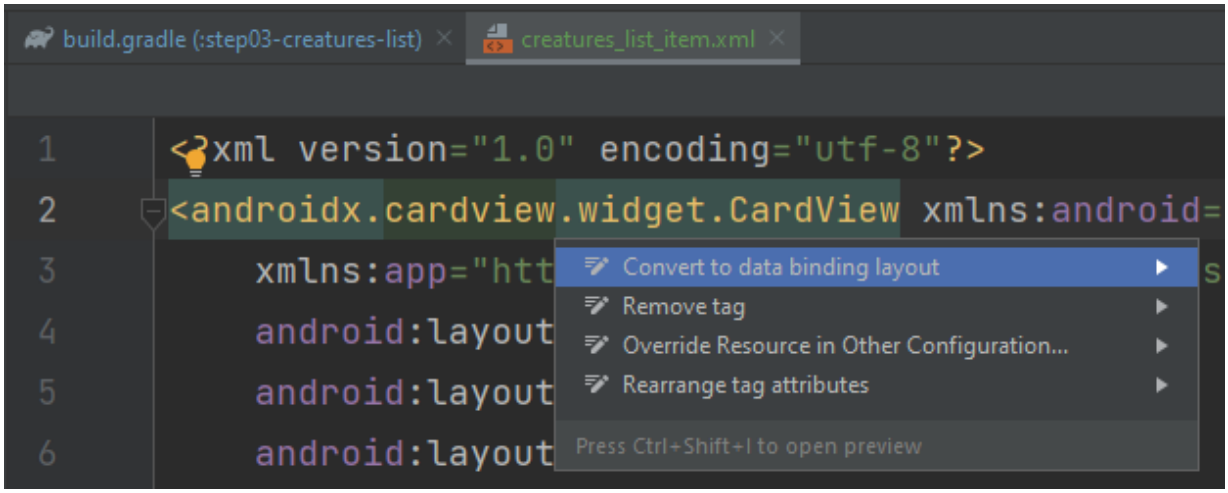


Figura 6.2: Converter XML para um layout com data binding.

Após aplicar a opção, o conteúdo do XML é alterado para a seguinte estrutura:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>
```

```
    </data>

    <androidx.cardview.widget.CardView ...>
        <!-- ... -->
    </androidx.cardview.widget.CardView>
</layout>
```

A partir de agora, esse arquivo de *layout* está com o *Data Binding* ativado. A tag `<layout>` passa a ser a raiz do arquivo, contendo informações dos *namespaces* do XML (`xmlns`) que estão sendo utilizados pelos elementos do arquivo. Dentro dessa tag, antes de iniciar as declarações das *Views*, temos a tag `data` , que é destinada às variáveis (`<variable>`) de dados que são utilizadas para obtenção de informações. Além disso, é possível adicionar tags importando determinados tipos (`<import>`) que são utilizados em situações específicas.

Criando variáveis de *Data Binding*

O caso mais comum de declaração dentro da tag `<data>` é a tag `<variable>` . Dentro de uma arquitetura MVVM, é mais comum ainda declararmos o `ViewModel` como uma variável do *layout*, de modo que os próprios elementos da *View* peguem informações diretamente dele. Nos capítulos mais à frente, declararemos o `ViewModel` dentro de um *layout*. Por ora, como estamos dentro do *layout* de cada item da lista, declaramos uma variável para armazenar o valor do item.

Para criar uma variável, precisamos de duas informações: o **nome da variável** e o **tipo da informação**. Neste caso, crie uma nova tag `<variable>` com o nome `creature` e o tipo `tech.salvatore.livro_android_kotlin_paulo_salvatore.model.domain.Creature` , da seguinte forma:

```
<!-- ... -->
<data>

    <variable
        name="creature"
```

```
type="tech.salvatore.livro_android_kotlin_paulo_salvatore.model.domain.Cre
ature" />
</data>
<!-- ... -->
```

Com essa declaração, é possível utilizar a variável `creature` dentro de qualquer elemento. No capítulo anterior, utilizamos a declaração `tvCreatureName.text = item.name` para alterar o nome da criatura no `TextView`. Para substituir essa lógica de apresentação pelo *Data Binding*, abra o arquivo `CreaturesListAdapter` e comente essa linha, como a seguir:

```
fun bindView(item: Creature) {
    val tvCreatureNumber = itemView.findViewById<TextView>
(R.id.tvCreatureNumber)
    val tvCreatureName = itemView.findViewById<TextView>
(R.id.tvCreatureName)

    tvCreatureNumber.text = item.number.toString()
    // tvCreatureName.text = item.name
}
```

Ao executar a aplicação, o nome das criaturas voltará ao valor padrão do `TextView`, que no caso é o nome da primeira criatura, o `Java`. Volte para o arquivo `creatures_list_item.xml` e procure pelo `TextView` que define o nome da criatura.

```
<TextView
    android:id="@+id/tvCreatureName"
    ...
    android:text="Java"
    ... />
```

Com o *Data Binding* ativado no *layout*, podemos utilizar variáveis e propriedades dessas variáveis utilizando a sintaxe `@{}` em qualquer atributo da *View*. Para alterar o texto do elemento para a propriedade `creature.name`, basta substituir todo o conteúdo por `@{creature.name}`, da seguinte forma:

```
<TextView
    android:id="@+id/tvCreatureName"
    ...
    android:text="@{creature.name}"
    ... />
```

Para garantir que essa declaração vai funcionar, precisamos definir o valor da variável `creature` no momento em que o *layout* é inflado. Para isso, volte para o arquivo `CreaturesListAdapter.kt` e vá até a função `onCreateViewHolder()`.

Todo *layout* que possui o *Data Binding* ativado gera uma classe de *binding* customizada utilizando o nome do arquivo XML como referência. No caso do arquivo `creatures_list_item.xml`, a classe é chamada de `CreaturesListItemBinding`. Para que o XML seja inflado corretamente, é necessário utilizar o método `inflate` dessa classe:

```
// ...
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.databinding.CreaturesL
istItemBinding

class CreaturesListAdapter(
    // ...
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ViewHolder {
        val layoutInflater = LayoutInflater.from(parent.context)

        val binding = CreaturesListItemBinding.inflate(layoutInflater,
parent, false)

        return ViewHolder(binding)
    }
    // ...
}
```

Caso a classe `CreaturesListItemBinding` não esteja sendo encontrada, você pode abrir a aba `Gradle`, no canto superior direito do Android Studio, clicar com o botão direito no módulo que você estiver usando e selecionar a opção `Reload Gradle Project`, como mostra a imagem a seguir.

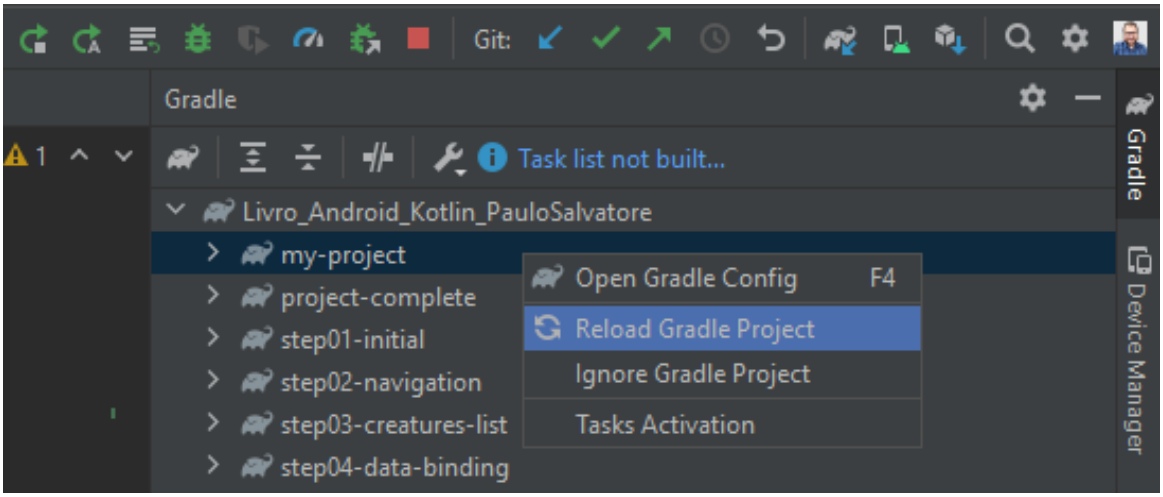


Figura 6.3: Reload Gradle Project.

Também é possível invalidar os caches da aplicação, com uma das seguintes formas:

- clicar no menu `Build | Make Project` (Ctrl + F9);
- no menu `Build | Clean Project`; ou
- no menu `Build | Rebuild Project`.

Além da alteração na função `onCreateViewHolder()`, é necessário alterar a classe `ViewHolder`, preparando-a para aceitar o `binding` do tipo `CreaturesListItemBinding`. Dessa forma, o `bindView` poderá definir o valor da variável `creature` diretamente. Com isso, também precisamos alterar a extensão `RecyclerView.ViewHolder()` para utilizar `binding.root` no lugar do `itemView` que não é mais passado.

```
inner class ViewHolder(
    private val binding: CreaturesListItemBinding
) : RecyclerView.ViewHolder(binding.root) {
```

```
    fun bindView(item: Creature) {
        binding.creature = item

        // ...
    }
}
```

Com essas alterações, execute o projeto (`Shift + F10`) e confirme se os nomes corretos das criaturas estão aparecendo, conforme estavam antes.

Variáveis numéricas

Entendendo o fluxo de implementação do *Data Binding*, vamos aplicar a mesma lógica para o número da criatura. Volte para o arquivo do *layout* e procure pelo `TextView` com o ID `tvCreatureNumber` . Altere o atributo `text` para `@{creature.number}` , como no trecho de código a seguir, e rode o projeto.

```
<TextView
    android:id="@+id/tvCreatureNumber"
    ...
    android:text="@{creature.number}"
    ... />
```

Ao executar a aplicação, recebemos o seguinte erro:

```
E/AndroidRuntime: FATAL EXCEPTION: main
Process: tech.salvatore.livro_android_kotlin_paulo_salvatore, PID:
8503
android.content.res.Resources$NotFoundException: **String resource ID
#0x1**
at android.content.res.Resources.getText(Resources.java:367)
at android.widget.TextView.setText(TextView.java:6370)
at tech.salvatore.livro_android_kotlin_paulo_salvatore.databinding.
CreaturesListItemBindingImpl.executeBindings(CreaturesListItemBindingI
mpl.java:124)
```

Quando estamos trabalhando com variáveis dentro de XML, é necessário tomar muito cuidado com os tipos das informações. Como a propriedade `Creature.number` é do tipo `Long`, ao tentar utilizá-la no atributo `text`, o Android entende isso como um recurso dentro do arquivo `strings.xml`, visto que cada recurso da classe `R` é tratado como um número inteiro, como é possível ver na imagem a seguir.

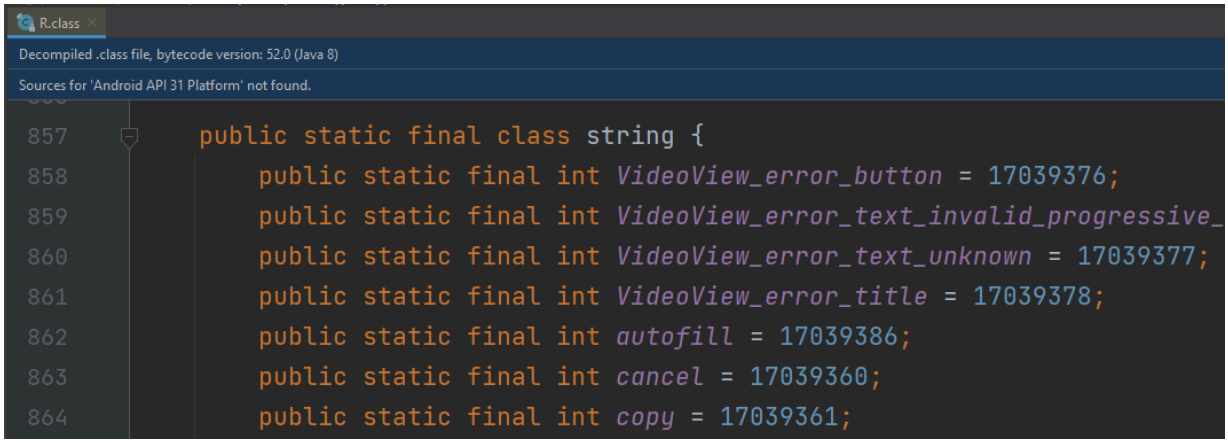


Figura 6.4: Conteúdo da classe R, em que recursos são registrados como números inteiros.

Por conta disso, quando queremos exibir números como texto, precisamos convertê-los para `string`, assim como já estava sendo feito na função `bindView()` da classe `CreaturesListAdapter`, no trecho a seguir:

```
// ...
fun bindView(item: Creature) {
    // ...
    tvCreatureNumber.text = item.number.toString()
}
// ...
```

Para converter uma informação direto no *layout* XML, devemos utilizar código Java, uma vez que não é possível utilizar código Kotlin. Portanto, volte para o arquivo `creatures_list_item.xml` e, no elemento `tvCreatureNumber`, adicione a declaração

`@{String.valueOf(creature.number)}` no atributo `text`, da seguinte forma:

```
<TextView
    android:id="@+id/tvCreatureNumber"
    ...
    android:text="@{String.valueOf(creature.number)}"
    ... />
```

Rode o projeto e confira se os números foram atualizados com sucesso.

Agora que temos o *Data Binding* exibindo o número da criatura, o código que realizava essa lógica de apresentação não é mais necessário. Abra o arquivo `CreaturesListAdapter.kt` para remover estes trechos, deixando como o código a seguir.

```
fun bindView(item: Creature) {
    binding.creature = item

    // Remover: val tvCreatureNumber = itemView.findViewById<TextView>
(R.id.tvCreatureNumber)
    // Remover: tvCreatureNumber.text = item.number.toString()
}
```

> Após remover os trechos, certifique-se de remover os *imports* que não são mais necessários. É possível utilizar o atalho `Ctrl + Alt + O` para otimizar os *imports*.

Converter valores numéricos para `strings` para conseguir exibi-los direto em elementos de *layout* é apenas uma das formas existentes. No entanto, quando é necessário adicionar mais algum texto na exibição, como o caso da criatura, em que o número `1` deve ser exibido como `# 001`, pode-se utilizar os recursos de `string`.

Novos recursos de `string` são declarados no arquivo `strings.xml`, dentro da pasta `res/values`. Em geral, esses recursos servem para organizar textos gerais da aplicação, principalmente para integrar

com o sistema de tradução presente no Android. No entanto, eles também permitem a utilização em conjunto com a sintaxe de formatação do Java, sendo possível combinar números com `strings`, além de formatar números.

Abra o arquivo `strings.xml` e, dentro da tag `<resources>`, adicione um recurso com a tag `<string>` com o *name* `creature_number` e o valor `# %1$03d`. A declaração `%1` será substituída pelo primeiro argumento que for passado. Enquanto a declaração `$d` significa que esse argumento deve ser numérico, `$03d` indica que esse número terá pelo menos 3 dígitos, inserindo o número 0 para preencher espaços faltantes. Dessa forma, o número `1` será substituído por `001`, assim como o número 99 será substituído por `099`. Por fim, adicionamos a *hashtag* (`#`) antes do número. Ao final da operação, teremos `# 001`.

```
<resources>
    <!-- ... -->
    <string name="creature_number"># %1$03d</string>
</resources>
```

Com o recurso criado, vamos substituir o atributo `text` do `TextView` para utilizá-lo, da seguinte forma:

```
<TextView
    android:id="@+id/tvCreatureNumber"
    ...
    android:text="@{@string/creature_number(creature.number)}"
    ... />
```

Rode o projeto (`Shift + F10`) e confira se a formatação foi aplicada com sucesso.

O próximo passo para deixar o *layout* de cada item completo é carregar as imagens de acordo com a URL de cada criatura.

6.2 Exibindo imagens

Trabalhar com o carregamento de imagens no Android parece uma tarefa bem simples a princípio — afinal, é apenas pegar uma imagem e exibi-la. O que poderia dar errado nisso? Geralmente é o que todos pensamos de início. A grande e dolorida verdade é que carregar imagens não é tão simples assim, independente de a imagem estar armazenada na memória do dispositivo ou em algum site da web.

O principal fator é o tamanho da imagem. Temos dispositivos hoje com câmeras cada vez mais poderosas que tiram fotos com resoluções altíssimas (mesmo que a tela do celular não reproduza a quantidade de pixels), e cujos arquivos não possuem tamanho tão grande. Como o dispositivo precisa armazenar essa informação na memória do celular (que é muito escassa), muitas vezes acaba resultando em uma `exception` conhecida como `OutOfMemory`.

Outro fator além do tamanho é quando a imagem está em um servidor da web, onde a comunicação dos dados deve ser feita via rede, o que, em muitos casos, é extremamente lento e pode apresentar interrupções e instabilidade.

Somando tudo isso, temos um fator inconsistente, que torna o carregamento de imagem uma tarefa complicada e demorada (em termos de execução). É por isso que o carregamento de imagens deve ser feito em processos separados, para não impactar diretamente o processo principal da aplicação, conhecido como `Main Thread` ou `UI Thread`.

Carregar grandes `bitmaps` de forma eficiente virou tópico de discussão na comunidade Android e pode ser feito quando conseguimos prever o tamanho da imagem antes de carregá-la, associando com o tamanho da `imageView` que receberá a informação, o que faz com que a imagem seja carregada em um tamanho menor. Isso acelera o processo e diminui o consumo de recursos, abrindo espaço também para definir diferentes tipos de configuração de

`bitmap` (como `ARGB_8888`, por exemplo) que alteram diretamente o consumo de bytes por pixel carregado.

No universo da programação, é muito comum nos depararmos com implementações frequentes com que todo projeto precisa lidar. Nesses casos, é muito comum existirem bibliotecas, geralmente de código aberto, que facilitam a inserção dessas funcionalidades em diferentes aplicações. Quando o assunto é carregamento de imagens, isso não é diferente. Atualmente, no Android, existem três principais bibliotecas que realizam essa função: **Glide**, da Bump Technologies; **Picasso**, da Square; e **Fresco**, do Facebook.

Quando precisamos decidir qual das bibliotecas utilizar, algumas questões precisam ser consideradas: sintaxe de utilização, tamanho da biblioteca (em KBs e quantidade de métodos), funcionamento padrão e funcionalidades extras disponíveis.

Sistema de cache das bibliotecas

Um dos pontos mais importantes a serem considerados é o sistema de cache que cada biblioteca utiliza, pois ele pode impactar diretamente a performance de uma aplicação.

O Glide conta com um sistema de cache baseado no tamanho da `ImageView`. Suponha que vamos renderizar uma imagem de 1000 x 1000 pixels em uma área de 100 x 100 pixels. Primeiro, ele faz o download da imagem 1000 x 1000 pixels e converte para 100 x 100, armazenando-a em um arquivo com tamanho 100 x 100 pixels no cache e carregando-a no `ImageView`.

Já o Picasso faz o cache da imagem no tamanho original, de 1000 x 1000 pixels, converte para 100 x 100 pixels e só depois a insere no `ImageView`.

A documentação do Fresco não informa se o sistema de cache é feito pelo tamanho do `ImageView` ou pelo tamanho da imagem, apesar de permitir uma gestão detalhada das imagens armazenadas no cache.

O resultado dessas diferenças de abordagem é que, geralmente, o Glide conseguirá carregar uma imagem mais rápido desde que ela já tenha sido carregada antes. No entanto, para que o Glide realmente seja mais rápido, o `ImageView` não pode ter um tamanho diferente de quando a imagem foi carregada pela primeira vez, pois nesse caso será necessário fazer o download novamente. Já o Picasso, mesmo com o cache da imagem, pode demorar mais, principalmente se a imagem original possuir um tamanho muito maior do que o `ImageView` em questão, pois sempre precisará fazer a conversão de tamanho antes de exibir.

No momento da escrita deste livro, um outro ponto que diferencia ambos é que o Glide e o Fresco suportam GIFs, enquanto o Picasso não. Portanto, se você deseja carregar esse tipo de imagem em sua aplicação, será necessário optar pelo Glide ou pelo Fresco.

Mais detalhes sobre a comparação entre as bibliotecas podem ser encontrados em alguns artigos pela internet, como este da Redwerk: https://redwerk.com/blog/advanced-feature-in-android-image-loaders-picasso-vs-glide-vs-fresco/

Figura 6.5: Artigo da Redwerk sobre a comparação de bibliotecas de imagem.

CRITERION	Picasso	glide	Fresco
Size	121 Kb/td>	440 Kb	500 kb
Convenience of use	high	high	middle
Web Download Speed	high	slightly lower due to lengthy caching processing	high
Cache Download Speed	average	high	high
Built-in Transformation Features	basic set of operations	basic set plus rounding	wide range of transformation capabilities, but with limitations
Additional tools	Absent.	Loading a frame from video as a picture and GIF, using any type of model, a flexible API with the ability to connect any network stack.	Saving images not in Java Heap, but in ashmem heap, the ability to crop images around any point, resize JPEG using native resources, support for Progressive JPEG images.

Figura 6.6: Comparação das bibliotecas de imagem.

Neste livro, utilizaremos a biblioteca Glide para o carregamento de imagem, por conta do sistema de cache, sintaxe de utilização, suporte a GIFs e tamanho total da aplicação.

Implementando o Glide

Para implementar o Glide no projeto, abra o arquivo do `build.gradle` do módulo que está utilizando e adicione as seguintes linhas dentro do bloco `dependencies` :

```
// ...
dependencies {
    // ...

    // Glide
    implementation("com.github.bumptech.glide:glide:$glideVersion")
    kapt("com.github.bumptech.glide:compiler:$glideVersion")
}
```

> Na maioria dos casos, a declaração do `kapt` não é necessária para que o Glide funcione. Ela só é necessária caso o projeto utilize as *annotations* presentes na biblioteca.

Como a aplicação carrega imagens da internet, é necessário sinalizar esse uso no manifesto da aplicação, através da declaração `<uses-permission>`. Abra o arquivo `AndroidManifest.xml` e insira o conteúdo a seguir no início do documento, antes da tag `<application>`, da seguinte forma:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="tech.salvatore.livro_android_kotlin_paulo_salvatore">

    <uses-permission android:name="android.permission.INTERNET" />

    <application ...>
    <!-- ... -->
```

Com o Glide devidamente configurado, vamos implementá-lo. A forma mais simples de utilizar o Glide é através da declaração a seguir. Obrigatoriamente devemos informar o contexto da aplicação (`.with(Context)`), uma URL da imagem (`.load(url)`) e a referência do ImageView (`.into(imageView)`).

```
Glide.with(this)
     .load(url)
     .into(imageView)
```

No nosso caso, o contexto pode ser a própria `ImageView`. A `url` é obtida através da propriedade `imageUrl` da criatura e a `ImageView` é obtida através da declaração `findViewById`, passando o ID `ivCreature`, como está declarado no arquivo de *layout* `creatures_list_item.xml`. Para exibir a imagem corretamente, adicione o trecho a seguir no arquivo `CreaturesListAdapter.kt`, dentro da função `bindView()`.

```
fun bindView(item: Creature) {
    // ...
    val ivCreature = itemView.findViewById<ImageView>(R.id.ivCreature)
    Glide.with(ivCreature).load(item.imageUrl).into(ivCreature)
}
```

Feito isso, rode a aplicação (`Shift + F10`) e veja o resultado.

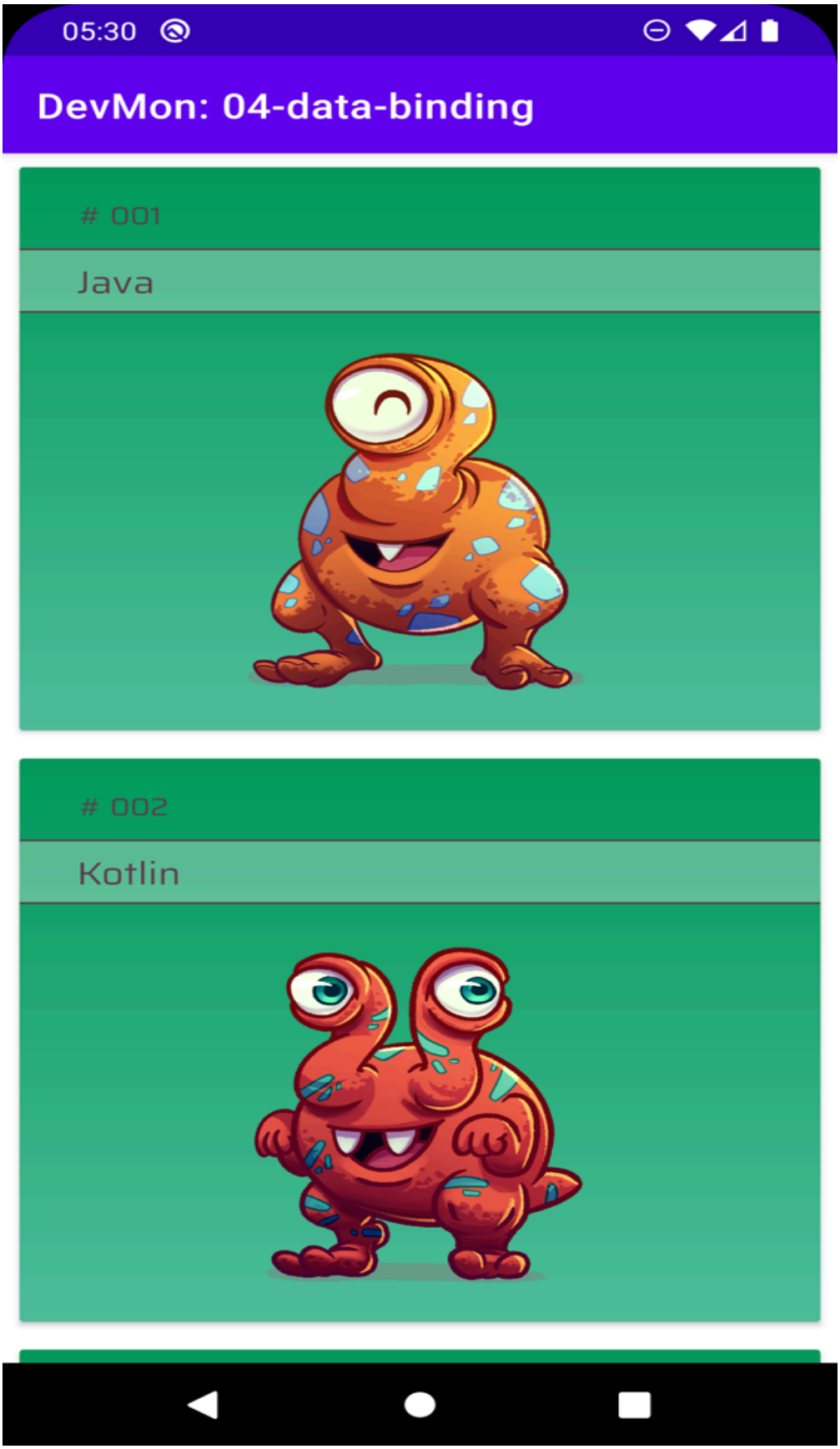

05:30
DevMon: 04-data-binding
001
Java
002
Kotlin

Figura 6.7: Resultado do carregamento de imagens.

No entanto, apesar de estar funcionando, dependendo da velocidade da conexão com a internet, é possível observar um comportamento estranho: inicialmente, a imagem da criatura Java aparece e só na sequência, após um tempo, a imagem é atualizada. Isso ocorre pois o elemento `ImageView` no *layout* possui um valor padrão para o atributo `android:src`.

Agora que estamos carregando a imagem da criatura pelo `bindView()`, podemos remover a declaração `android:src="@drawable/creature_java"` do arquivo de *layout*. Vá até o arquivo `creatures_list_item.xml`, busque pela `ImageView` e remova a declaração do `android:src`, como no código a seguir:

```
<ImageView
    android:id="@+id/ivCreature"
    android:layout_width="200dp"
    android:layout_height="200dp"
    android:layout_marginTop="20dp"
    android:layout_marginBottom="20dp"
    android:contentDescription="TODO: Content Description"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/barrier3" />
```

Com isso, toda a lógica de apresentação dos itens da lista foi movida para o recurso de *Data Binding* ou através de alguma lógica em Kotlin. Note que a prévia do *layout* no Android Studio acaba ficando vazia, como na imagem a seguir.

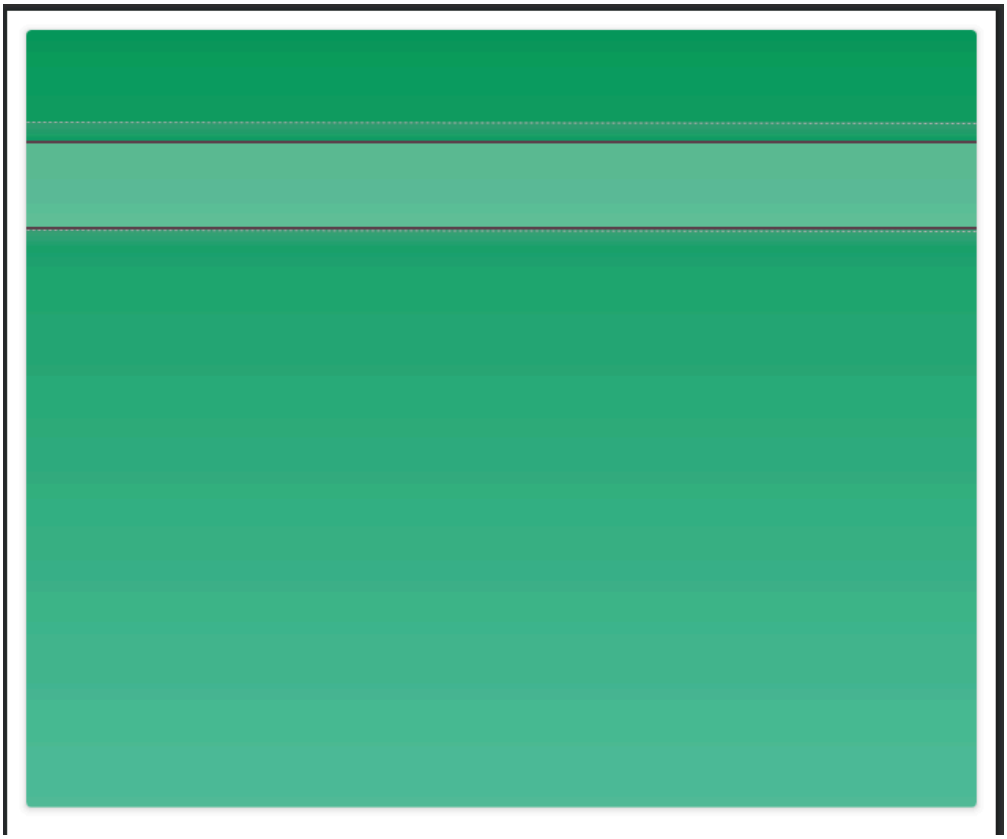


Figura 6.8: Prévia do layout no Android Studio.

Para melhorar essa visualização, podemos adicionar declarações do *namespace* `tools` nos elementos do *layout*. Procure pelos elementos do nome, número e imagem da criatura e adicione um valor que será utilizado apenas no Android Studio, como no trecho de código a seguir. Note que o `xmlns:tools` não vem importado por padrão nos arquivos de *layout*; é necessário referenciá-lo na tag `<layout>`.

```
<?xml version="1.0" encoding="utf-8"?>
<layout ...
    xmlns:tools="http://schemas.android.com/tools">

    <!-- ... -->

    <TextView
        android:id="@+id/tvCreatureNumber"
        ...
        tools:text="# 001" />
```

```
    <!-- ... -->

    <TextView
        android:id="@+id/tvCreatureName"
        ...
        tools:text="Java" />

    <!-- ... -->

    <ImageView
        android:id="@+id/ivCreature"
        ...
        tools:src="@drawable/creature_java" />

    <!-- ... -->
</layout>
```

Pronto! Agora temos uma prévia de *layout* com algumas informações de *placeholder*. Essas informações não são enviadas para o dispositivo, ficam apenas no Android Studio, como é possível ver na imagem a seguir.

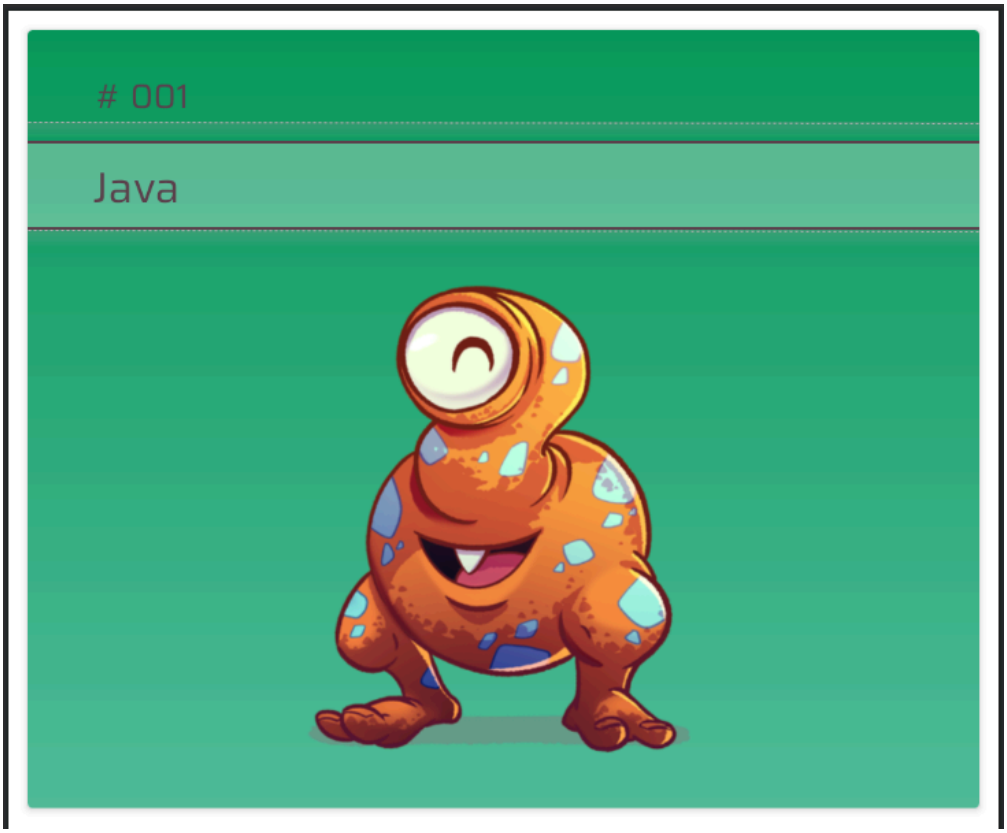


Figura 6.9: Prévia do layout no Android Studio com uso do tools.

Criando adapters customizados

Além de permitir utilizar variáveis dentro do *layout* XML, o *Data Binding* permite a criação de atributos customizados para as *Views*. Esses atributos são chamados de `Binding Adapters`.

Apesar de a declaração de carregamento de imagens com o Glide ser simples, tê-la declarada em diversos lugares dificulta a manutenção do código. Portanto, vamos criar um *binding adapter* customizado para abstrair essa lógica.

Como temos a URL da imagem associada a cada criatura através da declaração `creature.imageUrl`, podemos criar um atributo customizado chamado `app:imageUrl` que recebe essa URL através do *Data Binding* e efetua o carregamento da imagem.

Primeiro, vamos declarar no *layout* XML o novo atributo, informando a URL da imagem. Abra o arquivo `creatures_list_item.xml`, procure pela `ImageView` da criatura e adicione o seguinte conteúdo:

```
<ImageView
    android:id="@+id/ivCreature"
    app:imageUrl="@{creature.imageUrl}"
    ... />
```

Para que o `app:imageUrl` funcione, precisamos declará-lo em um arquivo Kotlin. Uma boa prática para a criação de *binding adapters* é a criação de um arquivo para cada tipo de *View*. No caso, vamos criar um arquivo para utilizar em *views* do tipo `ImageView`. Dentro do pacote `view`, crie um novo arquivo Kotlin do tipo `object` chamado `ImageBindingAdapter.kt`.

> O arquivo do *binding adapter* será do tipo `object`, pois não teremos instâncias da classe `ImageBindingAdapter`. A criação de objetos no Kotlin com esse tipo de declaração é transformada para uma classe com uma única instância no Java, representando um padrão conhecido como ***singleton***.

Dentro do arquivo, adicionamos uma nova função chamada `loadImage()` que fará o carregamento da imagens. Para isso, ela recebe dois parâmetros: a `ImageView` em que o atributo `app:imageUrl` está e a `url` da imagem, que corresponde ao texto que está declarado dentro das aspas `""` em `app:imageUrl="https://www.imagem.com/imagem.jpg"`. Dentro da função, verificamos se a URL possui uma informação válida e inicializamos o carregamento através do Glide, da seguinte forma:

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore.view

import android.widget.ImageView
import com.bumptech.glide.Glide

object ImageBindingAdapter {
```

```
    fun loadImage(view: ImageView, url: String?) {
        if (!url.isNullOrEmpty()) {
            Glide.with(view).load(url).into(view)
        }
    }
}
```

Para que o *Data Binding* entenda a função `loadImage()` e conecte com o atributo `app:imageUrl`, precisamos anotá-la com `@BindingAdapter()`. Além disso, somente funções que possuam o modificador `static`, do Java, são reconhecidas. Para que uma função em Kotlin tenha esse modificador durante a compilação, adicionaremos a *annotation* `@JvmStatic`:

```
object ImageBindingAdapter {
    @JvmStatic
    @BindingAdapter("imageUrl")
    fun loadImage(view: ImageView, url: String?) {
        // ...
```

> **Dica:** caso queira visualizar o código Java que é gerado a partir de um código Kotlin, vá até o menu `Tools | Kotlin | Show Kotlin Bytecode` e, na aba `Kotlin Bytecode` que foi aberta, clique no botão **`Decompile`**.

Para conferir se as declarações foram feitas corretamente, o código Java gerado deverá corresponder ao trecho a seguir.

```
// ...
public final class ImageBindingAdapter {
   @NotNull
   public static final ImageBindingAdapter INSTANCE;

   @JvmStatic
   @BindingAdapter({"imageUrl"})
   public static final void loadImage(@NotNull ImageView view, @Nullable
String url) {
       // ...
```

> O objeto e as funções declaradas no arquivo `ImageBindingAdapter.kt` serão utilizados apenas na geração de código do *Data Binding*; portanto, o Android Studio mostrará alguns avisos indicando que eles estão sem uso. Para ignorar esse aviso, adicione no começo do arquivo a declaração `@file:Suppress("unused")`.

Rode o projeto (`Shift + F10`) e teste o resultado.

Conclusão

Com o que vimos neste capítulo, foi possível perceber que *Data Binding* é uma biblioteca muito útil, principalmente para reduzir a quantidade necessária de escrita de código de lógica de apresentação para atualizar dados de *layouts* XML, removendo muitas declarações de `findViewById` desnecessárias.

Além disso, vimos como carregar imagens utilizando a biblioteca Glide e como criar atributos customizados de *views* no XML utilizando os *binding adapters*. Com eles, fizemos o atributo `app:image`, que recebe uma URL e carrega uma imagem em qualquer `ImageView`.

No próximo capítulo, implementaremos o sistema de esconder criaturas desconhecidas, mostrando apenas as criaturas pertencidas pelo usuário, sendo que um novo usuário será gerado sempre que o app fechar e abrir novamente.

Para finalizar este capítulo, chegou a hora de revelar que o nome do DevMon #7 é o **JavaScript**! Essa linguagem de programação que iniciou sem muita pretensão, apenas com o objetivo de ser uma linguagem simples para a execução de scripts direto no navegador, rapidamente conquistou diversos outros espaços a partir do lançamento Node.JS, que permite que a linguagem rode direto em servidores. Apesar de o nome possuir a palavra Java, as linguagens não são nada parecidas, nem são da mesma empresa. Nomeada

originalmente como Mocha e depois LiveScript, rapidamente mudou para JavaScript, pegando carona na popularidade que a linguagem Java estava ganhando na época.

URL da imagem: https://i.imgur.com/WEKXaw3.png

Figura 6.10: O DevMon #7 se chama JavaScript!

CAPÍTULO 7
Adquirindo criaturas

> Caso queira visualizar as implementações completas deste capítulo, acesse o módulo `CAP07-ADD-CREATURES`.

Na Pokédex original, toda a lista de Pokémon™ é uma incógnita. À medida que novas criaturas são encontradas pelo caminho, a Pokédex as identifica e exibe mais informações, além de salvar uma imagem dela. No DevMon não é diferente. Vamos construir esse comportamento, inicializando todas as criaturas mostrando apenas sua silhueta. A cada intervalo de tempo (que é configurável), um botão é liberado na tela principal para que uma nova criatura apareça. Apenas criaturas conhecidas terão seus nomes e imagens revelados.

No DevMon, assim que o aplicativo for iniciado pela primeira vez, vamos criar um registro de usuário e liberar uma primeira criatura para ser obtida. Para simular o comportamento de um usuário conectado, vamos criar uma classe `User` e gerenciar o *Model* na classe `UserRepository`, armazenando tudo em memória. Nessa classe, é possível guardar informações como o nome do usuário, se há criaturas disponíveis ou não e quais as criaturas que já foram obtidas.

Como as informações de usuário estão na memória, caso a aplicação seja fechada, um novo usuário é gerado e todo o comportamento é reiniciado. Posteriormente, é possível trabalhar o `UserRepository` para persistir essas informações no próprio dispositivo ou em algum servidor *back-end* na internet.

Assim que esse mecanismo estiver concluído, vamos implementar a evolução das criaturas, utilizando um sistema parecido, mas clicando em um botão na tela de cada criatura, que também é liberado após

um determinado tempo. Quando todas as criaturas forem obtidas, a criatura lendária aparecerá e poderá ser capturada.

Como de costume, vamos falar sobre a criatura que será revelada neste capítulo, o DevMon #8, nomeado em homenagem a uma linguagem de programação que vem ganhando cada vez mais atenção dentro do cenário JavaScript. É uma linguagem que, assim como o Kotlin, transpila códigos para uma outra linguagem. Já sabe qual é?

Figura 7.1: Qual é o DevMon #8?

7.1 Iniciando a sessão do usuário

Para começar, precisamos da entidade `User`, com uma estrutura que permita armazenar dados de usuário comuns como o nome, além de

duas informações cruciais para o fluxo de adição de criaturas e armazenamento das criaturas adicionadas. Navegue até o pacote principal da aplicação, na pasta `model/domain`, crie uma nova classe Kotlin com o nome `User.kt` com o conteúdo:

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore.model.domain

data class User(
    val name: String,

    val hasCreatureAvailable: Boolean,
    val creatures: List<Creature> = emptyList(),
)
```

Com a entidade criada, precisamos de uma classe de *Repository*, que chamaremos de `UserRepository`, responsável por guardar a referência do usuário da aplicação. Como mencionado anteriormente, a instância do usuário é armazenada na memória do dispositivo. Futuramente, quando quisermos manter as sessões, podemos alterar apenas o código da classe `UserRepository`.

Para criá-la, navegue até o pacote `model/repository` e crie uma nova classe Kotlin com o nome `UserRepository.kt`. A nova classe está no escopo `@Singleton` do Hilt, para que apenas uma única instância exista na aplicação, além de armazenar uma instância da classe `User` com um nome (pode ser qualquer nome de sua escolha) e indicar que há uma criatura disponível para ser coletada. Vamos ao código:

```
package
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.repository

import
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.domain.User
import javax.inject.Inject
import javax.inject.Singleton

@Singleton
class UserRepository @Inject constructor() {
```

```
    val user = User("Paulo Salvatore", true)
}
```

Com as duas classes criadas, finalizamos a construção, por enquanto, da camada *Model*. Para integrar com as *Activities* e *Fragments* da aplicação, precisamos também da camada *View Model*, que é responsável por expor as informações do *Repository*.

Vamos criar a classe `UserViewModel`, preparando-a para injeção com o Hilt e expondo o usuário recém-criado:

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore.viewmodel

import androidx.lifecycle.ViewModel
import dagger.hilt.android.lifecycle.HiltViewModel
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.domain.User
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.repository.UserR
epository
import javax.inject.Inject

@HiltViewModel
class UserViewModel @Inject constructor(
    userRepository: UserRepository,
) : ViewModel() {
    val user = userRepository.user
}
```

Para integrar a camada *View Model* na camada *View*, vamos adicionar a injeção do `UserViewModel` na `NavigationActivity` utilizando a declaração `by viewModels()`, como vimos nos capítulos anteriores. A `NavigationActivity` será responsável por identificar que a propriedade `hasCreatureAvailable` está com o valor `true` para alterar o `Fragment` que está sendo renderizado no momento, exibindo uma nova tela, que chamaremos de `CreatureChoose`, para que uma nova criatura seja escolhida. Portanto, abra a classe `NavigationActivity` e adicione o seguinte trecho logo após a propriedade `navigationViewModel`:

```
// ...
@AndroidEntryPoint
class NavigationActivity : AppCompatActivity() {
    // ...
    private val userViewModel: UserViewModel by viewModels()
    // ...
```

Agora que temos o `user` criado no `UserRepository` e disponibilizado para a *View* através do `UserViewModel`, podemos implementar o fluxo de escolha e adição de criaturas, como vimos no gráfico de navegação da aplicação.

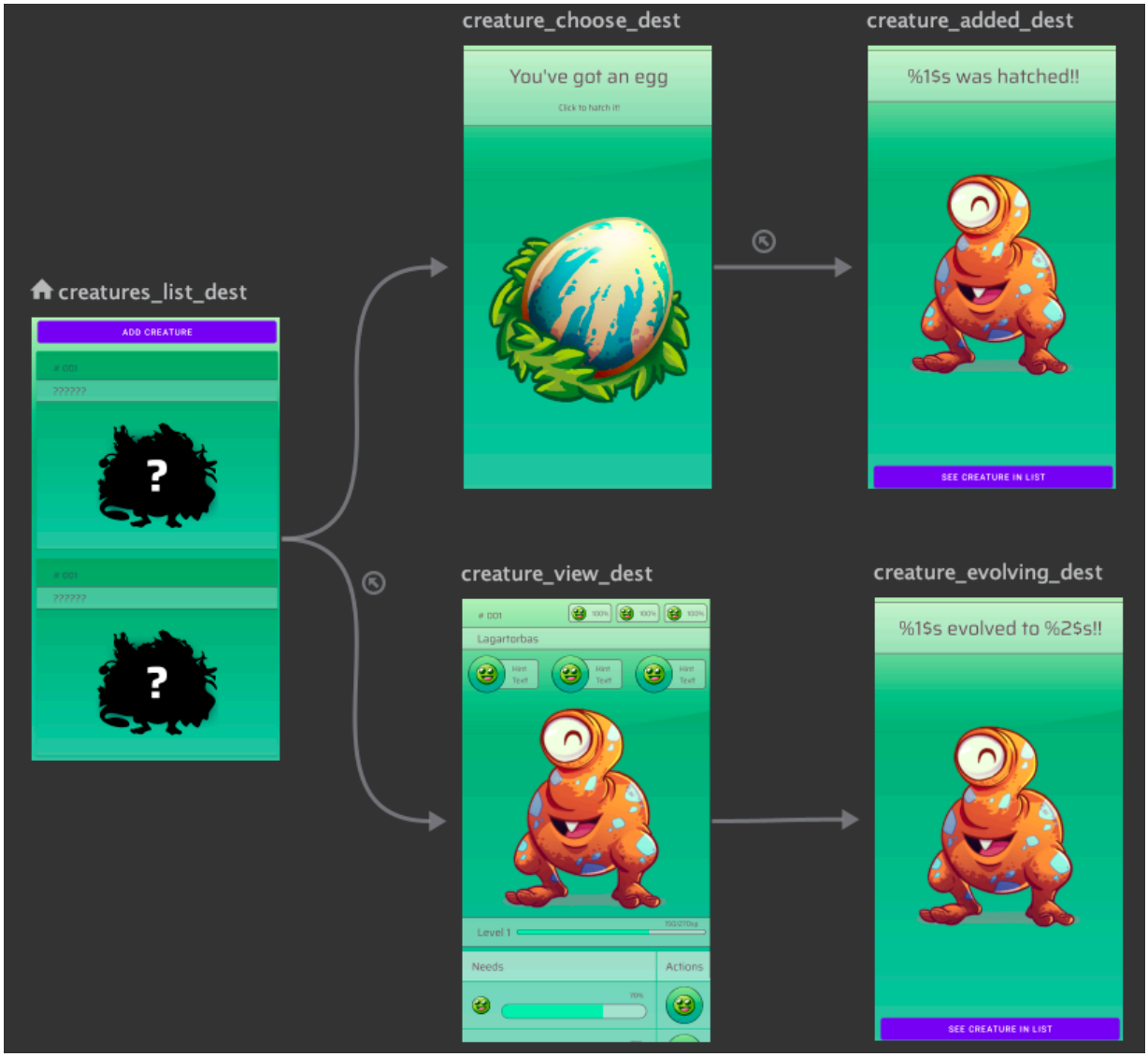


Figura 7.2: Fluxo de navegação completo da aplicação.

7.2 Criação da tela de escolher criatura

A tela de escolha de criatura é a primeira tela a aparecer quando o usuário possui a propriedade `hasCreatureAvailable` igual a `true`. Assim que a navegação é inicializada em `NavigationActivity.initNavigation()`, devemos checar o valor da propriedade e disparar a navegação utilizando a classe

`NavController`, disponível na biblioteca Navigation. Antes de criar as telas e inseri-las no fluxo de navegação, vamos organizar o pacote da `view`, criando os pacotes `binding_adapters`, `creatures`, `navigation` e organizando as classes existentes, da seguinte forma:

- Crie um novo pacote chamado `binding_adapters` e mova a classe `ImageBindingAdapter` para o pacote recém-criado;
- Crie outro pacote chamado `creatures` e mova as classes `CreaturesListAdapter` e `CreaturesListFragment` para ele;
- Crie um último pacote chamado `navigation` e mova a classe `NavigationActivity` para ele.

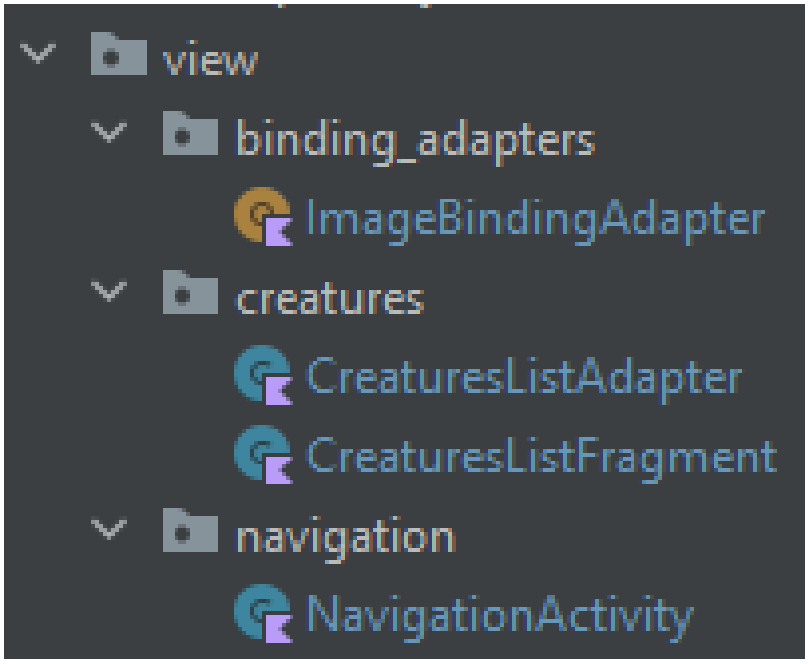


Figura 7.3: Estrutura de pacotes da view.

Agora que a camada *View* está organizada, vamos criar a tela `CreatureChooseFragment`, com o layout a seguir:

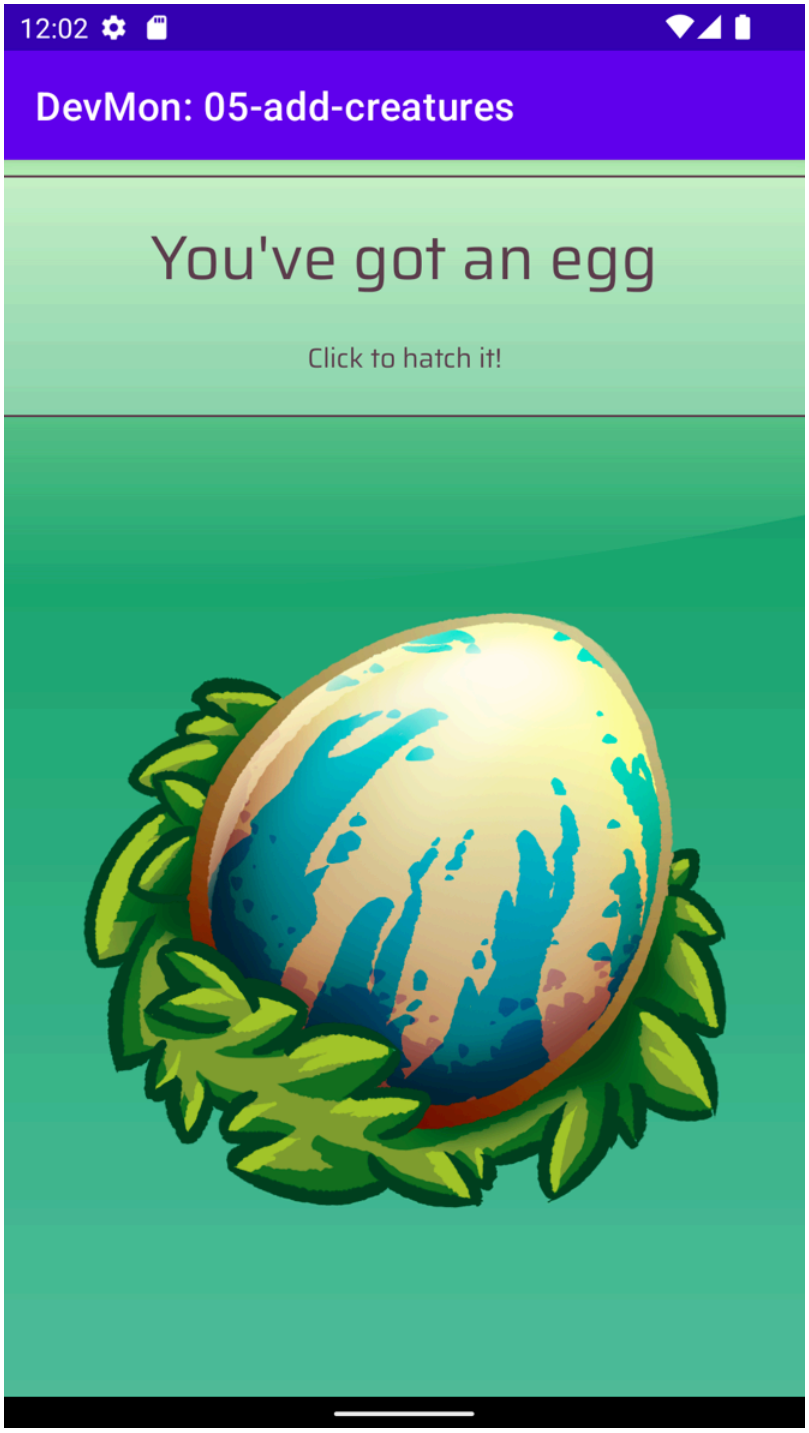


Figura 7.4: Layout da tela de escolher criatura.

Antes de criar o *Fragment*, vamos adicionar dois recursos visuais à pasta *drawable*: o detalhe de fundo no topo da tela e a imagem do ovo da criatura. Navegue até a pasta `project-complete/res/drawable` e

procure pelos arquivos `icon_egg_1.png` e `screen_top_background.xml`. Faça uma cópia deles e adicione à pasta `res/drawables` do seu app.

Com os recursos criados, vamos preparar o arquivo `strings.xml` com os textos que compõem a tela de escolha da criatura. Abra o arquivo `res/values.strings.xml` e adicione o seguinte trecho:

```
<resources>
    <!-- ... -->
    <string name="monsters_egg">Monster/'s egg</string>
    <string name="youve_got_an_egg">You/'ve got an egg</string>
    <string name="click_to_hatch_it">Click to hatch it!</string>
</resources>
```

Agora que os arquivos de recursos estão prontos, vamos criar o *Fragment* da tela. Clique com o botão direto no pacote `view/creatures` e selecione a opção `New | Kotlin Class/File` para criar uma nova classe chamada `CreatureChooseFragment.kt`. Depois, clique com o botão direito na pasta `res/layout` e selecione a opção `New | Layout Resource File` para criar um novo arquivo de *layout* chamado `creature_choose_fragment.xml`.

Fique à vontade para criar o *layout* da tela, de acordo com a referência demonstrada anteriormente. Além dos elementos visuais, o arquivo deverá possuir *Data Binding* ativado, referenciando uma variável chamada `viewModel` do tipo `UserViewModel`. Além disso, a imagem do ovo da criatura deverá ser um botão que, ao ser clicado, executa uma função do `UserViewModel` que vamos criar em breve.

O código XML na íntegra está disponibilizado a seguir:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">

    <data>

        <variable
```

```
            name="viewModel"

type="tech.salvatore.livro_android_kotlin_paulo_salvatore.viewmodel.UserVi
ewModel" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:id="@+id/main"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@drawable/screen_background"
        tools:context=".view.creatures.list.CreaturesChooseFragment">

        <ImageView
            android:id="@+id/ivBackgroundTop"
            android:layout_width="0dp"
            android:layout_height="221dp"
            android:scaleType="fitXY"
            android:src="@drawable/screen_top_background"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            tools:ignore="ContentDescription" />

        <androidx.constraintlayout.widget.ConstraintLayout
            android:id="@+id/constraintLayoutTitleSection"
            style="@style/creatures_background_title_section"
            android:layout_height="124dp"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent">

            <TextView
                android:id="@+id/tvTitle"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_marginTop="16dp"
                android:fontFamily="@font/saira_400"
                android:text="@string/youve_got_an_egg"
```

```
                android:textColor="#5D414F"
                android:textSize="32sp"
                app:layout_constraintEnd_toEndOf="parent"
                app:layout_constraintStart_toStartOf="parent"
                app:layout_constraintTop_toTopOf="parent" />

            <TextView
                android:id="@+id/tvSubtitle"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_marginTop="16dp"
                android:fontFamily="@font/saira_400"
                android:text="@string/click_to_hatch_it"
                android:textColor="#5D414F"
                android:textSize="14sp"
                app:layout_constraintEnd_toEndOf="parent"
                app:layout_constraintStart_toStartOf="parent"
                app:layout_constraintTop_toBottomOf="@id/tvTitle" />

        </androidx.constraintlayout.widget.ConstraintLayout>

        <ImageButton
            android:id="@+id/btChooseCreature"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:layout_weight="0.5"
            android:adjustViewBounds="true"
            android:background="@null"
            android:contentDescription="@string/monsters_egg"
            android:padding="32dp"
            android:scaleType="fitCenter"
            android:src="@drawable/icon_egg_1"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"

app:layout_constraintTop_toBottomOf="@id/constraintLayoutTitleSection" />
```

```
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Com o arquivo criado, certifique-se de clicar na opção `Make Project` para que o *Data Binding* gere o arquivo do *layout* que é necessário na classe do *Fragment*. Para integrar o fluxo de navegação com o *layout* recém-criado, vamos implementar a classe `CreatureChooseFragment` de forma bem simples, com o *boilerplate* padrão de *Fragments*, além de obter a instância do `UserViewModel` e adicionar ao *binding*, da seguinte forma:

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore.view.creatures

import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment
import androidx.fragment.app.viewModels
import dagger.hilt.android.AndroidEntryPoint
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.databinding.CreatureCh
ooseFragmentBinding
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.viewmodel.UserViewMode
l

@AndroidEntryPoint
class CreatureChooseFragment : Fragment() {
    private lateinit var binding: CreatureChooseFragmentBinding

    private val viewModel: UserViewModel by viewModels()

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        binding =
```

```
            CreatureChooseFragmentBinding.inflate(
                layoutInflater,
                container,
                false
            )

        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        binding.viewModel = viewModel
    }
}
```

Com a tela criada, vamos inseri-la no fluxo de navegação. Para isso, abra o arquivo `res/navigation/nav_graph.xml`. Assim como fizemos no capítulo *Fluxo de navegação*, clique no botão `New Destination` e selecione a tela `CreatureChooseFragment`. Altere o campo `id` para `creature_choose_dest` e o campo `label` para `creature_choose_fragment`. Para adicionar um *preview* para a tela, vá até o código XML do `nav_graph` e adicione a propriedade `tools:layout` à tag `<fragment>` referenciando o *layout* correto, como mostra o trecho a seguir.

```
<fragment
    android:id="@+id/creature_choose_dest"

android:name="tech.salvatore.livro_android_kotlin_paulo_salvatore.view.cre
atures.CreatureChooseFragment"
    android:label="creature_choose_fragment"
    tools:layout="@layout/creature_choose_fragment" />
```

Agora, para conectar a tela de lista de criaturas com a tela de escolha de criatura, volte para a visão de `Design` e coloque o mouse em cima da tela de lista de criaturas. No círculo que fica no canto direito da tela, clique, segure e arraste em cima da tela de escolha de criatura, como mostra a imagem a seguir.

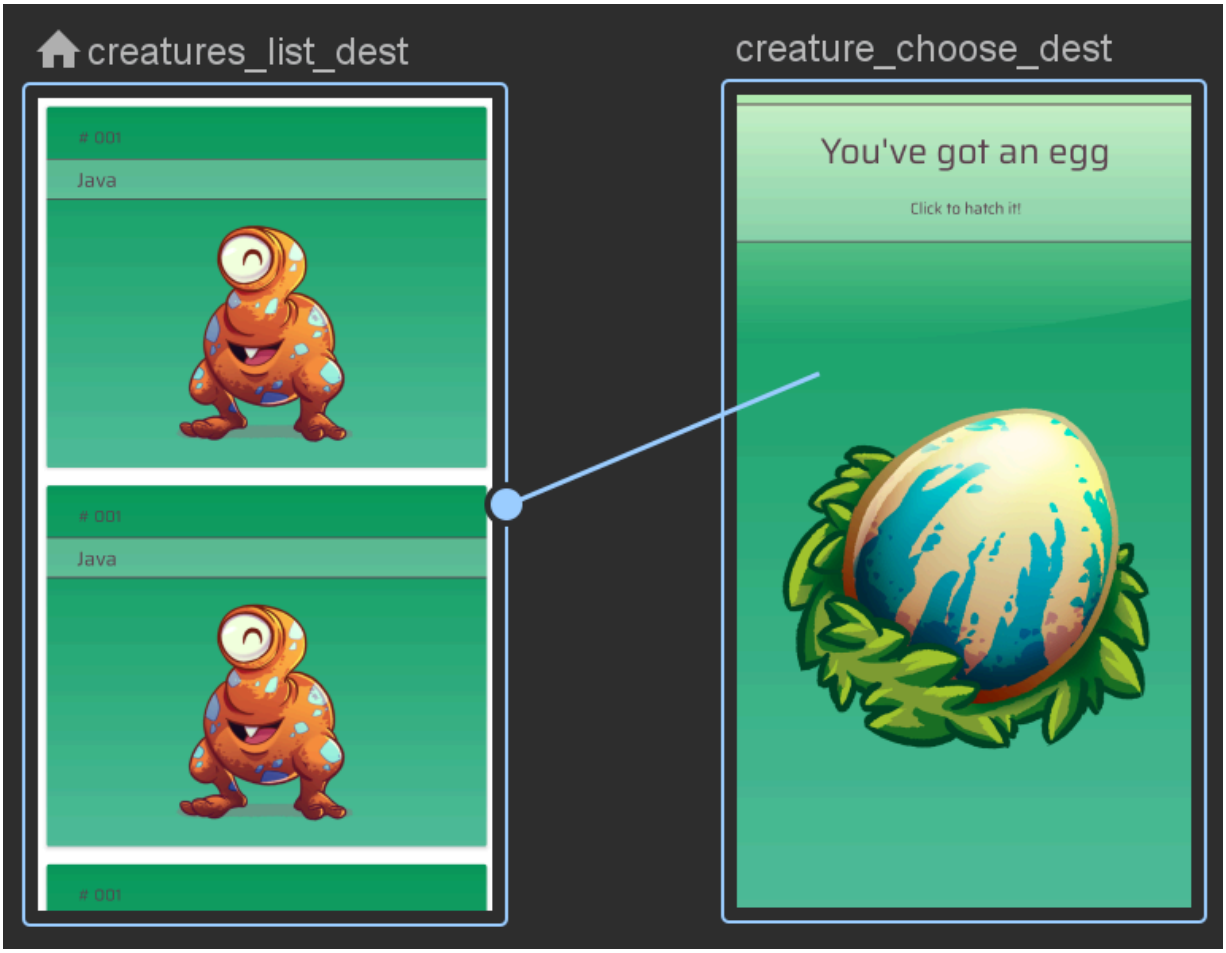


Figura 7.5: Conectando lista com escolha de criatura.

Com isso, uma **action** é criada e pode ser utilizada para navegar de uma tela para a outra. Altere o `id` da *action* para `creature_choose_action`. O arquivo `nav_graph.xml` deverá conter o seguinte código:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph.xml"
    app:startDestination="@id/creatures_list_dest">

    <fragment
        android:id="@+id/creatures_list_dest"
```

```
android:name="tech.salvatore.livro_android_kotlin_paulo_salvatore.view.creatures.CreaturesListFragment"
        android:label="creatures_list_fragment"
        tools:layout="@layout/creatures_list_fragment" >
        <action
            android:id="@+id/creature_choose_action"
            app:destination="@id/creature_choose_dest" />
    </fragment>

    <fragment
        android:id="@+id/creature_choose_dest"

android:name="tech.salvatore.livro_android_kotlin_paulo_salvatore.view.creatures.CreatureChooseFragment"
        android:label="creature_choose_fragment"
        tools:layout="@layout/creature_choose_fragment" />
</navigation>
```

> Sempre que criar *actions* de navegação, certifique-se de clicar no botão `Make Project` para que o Android gere os arquivos de código do navigation.

Com a *action* criada, vamos dispará-la sempre que a aplicação iniciar e o usuário tiver uma criatura disponível para escolher. Abra a classe `NavigationActivity` e vá até o método `initNavigation()`. Logo após a configuração do `NavGraph`, vamos adicionar o trecho de código que faz a checagem e verificar se a tela atual do navigation é diferente da tela de escolha de criatura. Caso a checagem passe, vamos pegar a `action` e realizar o disparo através da declaração `navController.navigate(action)`:

```
// ...
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.view.creatures.CreaturesListFragmentDirections
// ...
```

```
class NavigationActivity : AppCompatActivity() {
    // ...
    private fun initNavigation() {
        // Add NavGraph
        val graphInflater = navController.navInflater
        val navGraph = graphInflater.inflate(R.navigation.nav_graph)
        navController.graph = navGraph

        if (userViewModel.user.hasCreatureAvailable
            && navController.currentDestination?.id !=
R.id.creature_choose_dest
        ) {
            val action =
CreaturesListFragmentDirections.creatureChooseAction()
            navController.navigate(action)
        }
    }
}
```

Para testar se está tudo funcionando, rode a aplicação e confirme que a primeira tela que abre é a tela de escolha de criatura, exibindo a imagem do ovo da criatura.

Para finalizar o funcionamento da tela, precisamos implementar o comportamento que acontece ao clicar no ovo. Para isso, vamos criar um método chamado `chooseCreature()` dentro do `UserRepository` que checa se a propriedade `hasCreatureAvailable` do `User` é igual a `true` e adiciona uma criatura aleatória à lista de criaturas do usuário. Veja o código:

```
// ...
class UserRepository @Inject constructor(
    // ...
    fun chooseCreature() {
        if (!user.hasCreatureAvailable) {
            return
        }

        user.hasCreatureAvailable = false
```

```
        val randomCreature = creaturesRepository.creatures.random()
        user.creatures.add(randomCreature)
    }
}
```

Para conseguir executar o método que acabamos de criar no *Repository*, precisamos que ele esteja exposto através do *View Model*. Portanto, abra o arquivo `UserViewModel` e crie um método chamado `chooseCreature()` que executa o método `userRepository.chooseCreature()`:

```
// ...
class UserViewModel @Inject constructor(
    // ...
    fun chooseCreature() {
        userRepository.chooseCreature()
    }
}
```

Por fim, vamos fazer com que, ao clicar no ovo, seja disparado o método `chooseCreature()` do `UserViewModel`. Executar métodos diretamente dos arquivos de *layout* é possível com o uso do *Data Binding*, através da declaração `() -> metodo()` dentro de propriedades de eventos como o `onClick`. Abra o arquivo `creature_choose_fragment.xml`, procure pelo elemento `btChooseCreature` e adicione o trecho a seguir.

```
<ImageButton
    android:id="@+id/btChooseCreature"
    ...
    android:onClick="@{() -> viewModel.chooseCreature()}"
    ... />
```

Pronto! Com isso, ao clicar na imagem do ovo, uma criatura aleatória será adicionada ao usuário e o valor da propriedade `hasCreatureAvailable` será alterado para `false`.

No entanto, após realizar toda a operação, a tela de escolha de criatura continua visível. Precisamos de uma forma de forçar a navegação para uma outra tela. Como a operação está sendo realizada na classe `UserRepository`, não temos como executar diretamente uma implementação da `NavigationActivity`, pois isso implicaria em a camada *Repository* ter conhecimento da camada *View*, o que acarretaria problemas de manutenção. Nesse tipo de situação, é comum utilizar um mecanismo orientado a eventos, como o que fizemos com o `LiveData` na implementação da propriedade `creatures` na classe `CreaturesViewModel`.

A utilização de eventos que avisam que determinado comportamento aconteceu é extremamente útil em diversas situações, pois, além de desacoplar camadas de aplicação, permite uma extensão bem poderosa ao longo da evolução de um software. Neste caso, vamos implementar um evento chamado `onChooseCreature`, que avisa para qualquer interessado que uma criatura acabou de ser escolhida. No caso da classe `UserRepository`, não importa o que será feito com essa informação, mas qualquer um que precisar dela poderá simplesmente observar a propriedade `onChooseCreature` e passar a receber a `Creature` que acabou de ser escolhida.

Um ponto de atenção aqui é que durante os capítulos anteriores vimos apenas esse tipo de utilização através da classe `LiveData`. Apesar de simples e poderosa, a classe `LiveData` precisa de um `LifecycleOwner` para permitir observações nos valores. Esse `LifecycleOwner` está presente apenas no caso de *views* que são *Activities* ou *Fragments*. Para as classes de *Repository*, que são acessadas por outras classes da mesma camada ou da camada *View Model*, isso não é uma realidade; portanto, não devemos utilizar `LiveData` em camadas além da *View Model*.

Outras bibliotecas oferecem suporte a esse tipo de implementação, o que é o caso da biblioteca *ReactiveX*, ou simplesmente **RX**. A utilização do RX não depende de `LifecycleOwner`, o que facilita sua utilização em outras camadas além da *View*.

No entanto, para acelerar o processo de implementação da aquisição de novas criaturas, vamos deixar a implementação do RX para o próximo capítulo. A não utilização do `LiveData` na camada *Repository* ocorre quando precisamos trabalhar com o valor armazenado, pois ao solicitá-lo com a declaração `liveData.observe()`, precisamos passar o `LifecycleOwner`, que não existe na camada *Repository*. Como este não é o caso, pois precisamos apenas do `LiveData` para expor que o evento de `onChooseCreature` ocorreu, podemos utilizá-lo sem que uma declaração de `observe()` seja necessária fora da *View*.

Vamos começar criando a propriedade `LiveData` na classe `UserRepository`, seguindo o padrão apresentado nos capítulos anteriores, em que utilizamos um `MutableLiveData` interno, expondo apenas o `LiveData` para utilização fora da classe. Além disso, o `LiveData` vai armazenar um dado do tipo `Creature`, que representa a criatura que acabou de ser escolhida:

```
// ...
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.domain.Creature

// ...
class UserRepository @Inject constructor(
    // ...
    private val _onChooseCreature = MutableLiveData<Creature>()
    val onChooseCreature: LiveData<Creature>
        get() = _onChooseCreature
    // ...
}
```

Após a construção do `onChooseCreature`, vamos garantir que ele seja executado logo que uma criatura é escolhida. Para isso, dentro do método `chooseCreature()`, logo que toda a operação acontece, atualizamos a propriedade `value` do `LiveData` com a criatura aleatória que foi escolhida.

```
// ...
class UserRepository @Inject constructor(
    // ...
    fun chooseCreature() {
        // ...
        _onChooseCreature.value = randomCreature
    }
}
```

Agora que o `UserRepository` está pronto, vamos deixar a propriedade `onChooseCreature` exposta para que a *View* consiga receber as atualizações devidamente. Abra o arquivo `UserViewModel` e adicione a seguinte propriedade logo após a declaração de `user`:

```
// ...
class UserViewModel @Inject constructor(
    // ...
    val onChooseCreature = userRepository.onChooseCreature
    // ...
}
```

Para observar as atualizações na propriedade `onChooseCreature`, vamos adicionar a declaração `observe()` na classe `NavigationActivity` logo após a execução do método `initNavigation()`, dentro do método `onCreate()`:

```
// ...
class NavigationActivity : AppCompatActivity() {
    // ...
    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
        userViewModel.onChooseCreature.observe(this) {
            // Uma criatura acabou de ser escolhida
        }
    }
}
```

Prontinho! Com essa implementação, conseguimos garantir que sempre que uma criatura for escolhida, esse bloco de código será

executado. O próximo passo é criar e exibir uma tela da criatura adicionada.

7.3 Criação da tela da criatura adicionada

Assim que a criatura for adicionada, vamos abrir uma nova tela para exibi-la. Essa tela será chamada de `CreatureAddedFragment` e terá o *layout* a seguir:

Figura 7.6: Tela de criatura adicionada.

Para implementar essa tela, seguiremos o mesmo fluxo feito na criação da tela de escolha da criatura: criação do arquivo do `Fragment` ; adição de `strings` ; criação do *layout* com *Data Binding*;

adição do *layout* ao fluxo de navegação e criação da *action* de navegação.

Dentro do pacote `view/creatures`, crie um arquivo chamado `CreatureAddedFragment.kt`. Abra o arquivo `res/values/strings.xml` e adicione as `strings` a seguir.

```
<resources>
    <!-- ... -->
    <string name="creature_hatched">%1$s was hatched!!</string>
    <string name="creatures_image">%1$s/'s Image</string>
    <string name="see_creature_in_list">See creature in list</string>
</resources>
```

No pacote `res/layout`, crie um arquivo chamado `creature_added_fragment.xml` e adicione o conteúdo a seguir.

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">

    <data></data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:id="@+id/main"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@drawable/screen_background"
        tools:context=".view.creatures.CreatureAddedFragment">

        <ImageView
            android:id="@+id/ivBackgroundTop"
            android:layout_width="0dp"
            android:layout_height="221dp"
            android:scaleType="fitXY"
            android:src="@drawable/screen_top_background"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent"
```

```
    tools:ignore="ContentDescription" />

<androidx.constraintlayout.widget.ConstraintLayout
    android:id="@+id/constraintLayoutTitleSection"
    style="@style/creatures_background_title_section"
    android:layout_height="84dp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent">

    <TextView
        android:id="@+id/tvTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:ellipsize="end"
        android:fontFamily="@font/saira_400"
        android:gravity="center"
        android:maxLines="1"
        android:textColor="#5D414F"
        android:textSize="32sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="@string/creature_hatched" />

</androidx.constraintlayout.widget.ConstraintLayout>

<ImageView
    android:id="@+id/ivCreature"
    android:layout_width="340dp"
    android:layout_height="340dp"
    android:layout_marginTop="20dp"
    android:contentDescription="@string/creatures_image"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:src="@drawable/creature_java" />
```

```
        <Button
            android:id="@+id/btAddCreature"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_marginStart="10dp"
            android:layout_marginLeft="10dp"
            android:layout_marginEnd="10dp"
            android:layout_marginRight="10dp"
            android:text="@string/see_creature_in_list"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent" />

    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Com o arquivo de *layout* criado com o *Data Binding* ativado, clique no botão `Make Project` para gerar os arquivos de `Binding`. Abra o arquivo `CreatureAddedFragment` e adicione o código *boilerplate* a seguir.

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore.view.creatures

import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment
import dagger.hilt.android.AndroidEntryPoint
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.databinding.CreatureAd
dedFragmentBinding

@AndroidEntryPoint
class CreatureAddedFragment : Fragment() {
    private lateinit var binding: CreatureAddedFragmentBinding

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
```

```
        savedInstanceState: Bundle?
    ): View {
        binding =
            CreatureAddedFragmentBinding.inflate(
                layoutInflater,
                container,
                false
            )

        return binding.root
    }
}
```

Seguindo o mesmo fluxo de adição de uma tela ao `nav_graph` que fizemos anteriormente com a tela de escolha de criatura, abra o arquivo `res/navigation/nav_graph.xml` e adicione a tela de criatura adicionada ao fluxo de navegação, renomeando o `id` do `<fragment>` para `creature_added_dest`, a `label` para `creature_added_fragment` e adicionando um *preview* do layout através da declaração `tools:layout`. Certifique-se também de adicionar uma *action* entre a tela `CreatureChooseFragment` e a tela `CreatureAddedFragment`, renomeando o `id` da *action* para `creature_choose_to_creature_added_action`. O código do `nav_graph.xml` deverá ficar da seguinte maneira:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation ...>
    <!-- ... -->
    <fragment
        android:id="@+id/creature_choose_dest"
        ...>
        <action
            android:id="@+id/creature_choose_to_creature_added_action"
            app:destination="@id/creature_added_dest" />
    </fragment>

    <fragment
        android:id="@+id/creature_added_dest"
```

```
android:name="tech.salvatore.livro_android_kotlin_paulo_salvatore.view.creatures.CreatureAddedFragment"
        android:label="creature_added_fragment" />
</navigation>
```

Um recurso bem interessante que está presente no Android Navigation é o envio de argumentos entre telas. No caso da tela de criatura adicionada, podemos enviar o `creatureNumber` da criatura que foi adicionada aleatoriamente na tela de escolha de criatura. Para fazer isso, na visualização de `Design` do gráfico de navegação, selecione a tela `creature_added_dest` e, na aba `Attributes`, procure pela seção `Arguments` e clique no botão `+`. Nomeie o novo argumento como `creatureNumber`, marque o tipo `Integer` e clique em `Add`, como mostra a figura a seguir.

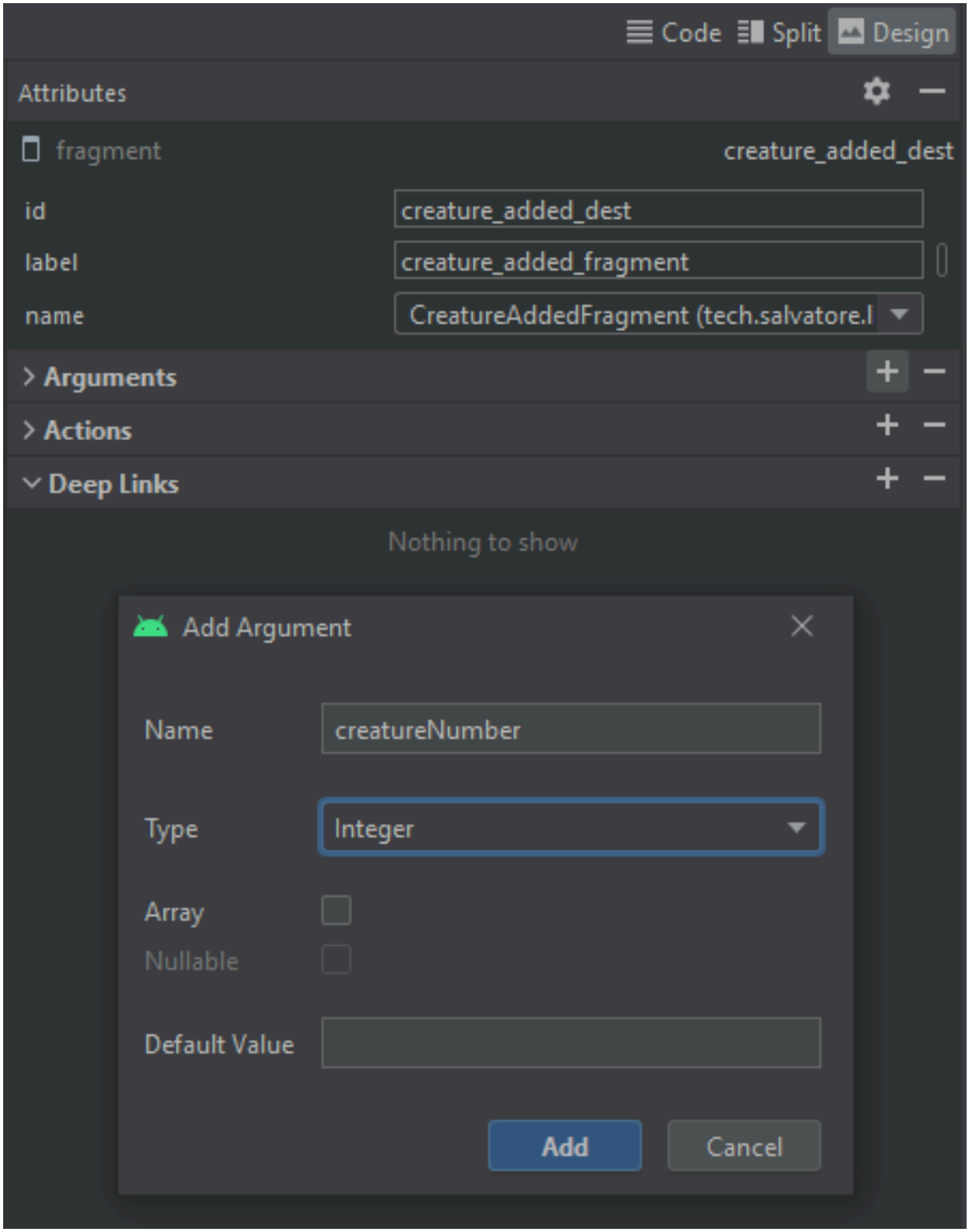


Figura 7.7: Adicionar argumento ao Navigation.

Com isso, o código do `nav_graph.xml` deve ficar como o trecho a seguir:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation ...>
    <fragment
        android:id="@+id/creature_added_dest"
        ...>
        <argument
            android:name="creatureNumber"
            app:argType="integer" />
    </fragment>
</navigation>
```

Clique em `Make Project` para garantir que o código da navegação será gerado corretamente.

Com a tela criada e inserida no fluxo de navegação, vamos voltar à classe `NavigationActivity.kt` para implementar a navegação assim que uma criatura for escolhida. Na declaração `userViewModel.onChooseCreature.observe`, precisamos primeiro checar se estamos na tela `creature_choose_dest` antes de navegar para a próxima tela através da *action* `creatureChooseToCreatureAddedAction`. Após nos certificarmos de que estamos na tela correta, disparamos a *action* informando o `number` da criatura que acabou de ser escolhida, da seguinte forma:

```
// ...
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.view.creatures.Creatur
eChooseFragmentDirections
// ...
class NavigationActivity : AppCompatActivity() {
    // ...
    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
        userViewModel.onChooseCreature.observe(this) {
            if (navController.currentDestination?.id ==
R.id.creature_choose_dest) {
```

```
                val action =

CreatureChooseFragmentDirections.creatureChooseToCreatureAddedAction(it.nu
mber)
                navController.navigate(action)
            }
        }
    }
    // ...
}
```

Com isso, ao clicar no ovo na tela de escolher criatura, imediatamente a tela de criatura adicionada é criada. Rode a aplicação para conferir se o comportamento está como o esperado.

> Note que a tela de criatura escolhida ainda não exibe qual a criatura recebida da tela anterior, pois ainda não implementamos esse funcionamento.

Figura 7.8: Tela de criatura adicionada.

Obtendo a criatura adicionada

Anteriormente, implementamos o envio do número da criatura como argumento entre as telas de escolher criaturas e de criatura adicionada. Para saber qual criatura devemos exibir, é necessário pegar o argumento e carregar na lista de criaturas disponíveis no `CreaturesViewModel`.

Primeiro, vamos obter o `creatureNumber` recebido como argumento no *Fragment* `CreatureAddedFragment`. O método equivalente ao `Activity.onCreate()` é o `Fragment.onViewCreated()`; portanto, vamos sobrescrever esse método e obter os argumentos do *navigation*.

Como o *navigation* atua com uma *lib* que gera código, com os argumentos de *actions* não seria diferente, uma vez que eles geram as classes com sufixo `LayoutNameFragmentArgs`, que, neste caso, é a classe `CreatureAddedFragmentArgs`. Além disso, para obter uma instância dessa classe, é possível utilizar a declaração `by navArgs()`. Com a instância obtida, basta acessar a variável que foi gerada para cada argumento:

```
// ...
class CreatureAddedFragment : Fragment() {
    // ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        val safeArgs: CreatureAddedFragmentArgs by navArgs()
        val creatureNumber = safeArgs.creatureNumber
    }
}
```

Agora que temos o `creatureNumber` disponível, vamos criar um método na camada *Repository* para buscar uma criatura pelo número. Abra o arquivo `CreaturesRepository.kt` e adicione o seguinte trecho:

```
// ...
class CreaturesRepository @Inject constructor() {
    // ...
    fun findCreature(number: Int) =
```

```
        creatures.find { it.number == number }
}
```

Vamos deixá-lo exposto no `CreaturesViewModel` para que possa ser utilizado pela *View*:

```
// ...
class CreaturesViewModel @Inject constructor(
    private val creaturesRepository: CreaturesRepository
) : ViewModel() {
    // ...
    fun findCreature(number: Int) =
creaturesRepository.findCreature(number)
}
```

> Note que, para acessar `creaturesRepository` dentro do método `findCreature`, é necessário transformar o parâmetro do construtor em uma propriedade, através da declaração `val`. Ao fazer isso, a propriedade se torna pública automaticamente. Vale ressaltar que não é uma boa prática deixar o acesso ao *Repository* diretamente exposto: é o *View Model* que deve intermediar todas as chamadas. Portanto, devemos adicionar o modificador de acesso `private`.

Agora que implementamos o método de busca, podemos usar o `creatureNumber` para carregar a criatura e enviar para o *layout*. Vamos preparar o *Data Binding* do *layout* para receber uma variável de criatura. Abra o arquivo `creature_added_fragment.xml` e adicione o seguinte trecho:

```
<?xml version="1.0" encoding="utf-8"?>
<layout ...>
    <data>
        <variable
            name="creature"

type="tech.salvatore.livro_android_kotlin_paulo_salvatore.model.domain.Cre
```

```
ature" />
    </data>
    <!-- ... -->
</layout>
```

Com essa declaração, o *layout* pode esperar que essa informação será passada em algum momento e passar a utilizá-la direto nos elementos de visualização. Vamos atualizar o `TextView tvTitle` para exibir o nome da criatura, formatando de acordo com a `string creature_hatched` e a `ImageView ivCreature` para exibir a imagem da criatura utilizando o *binding adapter* que criamos anteriormente, da seguinte forma:

```
<!-- ... -->
<androidx.constraintlayout.widget.ConstraintLayout ...>
    <TextView
        android:id="@+id/tvTitle"
        ...
        android:text="@{@string/creature_hatched(creature.name)}" />
</androidx.constraintlayout.widget.ConstraintLayout>

<ImageView
    android:id="@+id/ivCreature"
    ...
    app:imageUrl="@{creature.imageUrl}" />
<!-- ... -->
```

Com isso, clique em `Make Project` para que o código do *Data Binding* seja gerado corretamente.

> **Dica:** sempre que estiver trabalhando com bibliotecas que geram código, elas nem sempre conseguem gerar o código esperado, ou não atualizam quando deveriam. Uma dica para contornar esse problema é selecionar a opção `Build | Clean Project` em seguida de `Build | Make Project`.

Na classe `CreatureAddedFragment.kt`, vamos executar o método `findCreature()` do *View Model* e inserir a criatura encontrada direto no *layout Data Binding*:

```
// ...
class CreatureAddedFragment : Fragment() {

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        // ...
        // Find Creature
        val creature = viewModel.findCreature(creatureId)
        binding.creature = creature
    }
}
```

Por fim, vamos aproveitar para implementar o botão `See creature in list`, que deve retornar para a lista de criaturas. Para implementar o mecanismo de voltar a uma tela, precisamos entender como o *navigation* lida com isso. Ao analisar o gráfico de navegação, como na figura a seguir, temos três telas; a lista de criaturas é o ponto de partida.

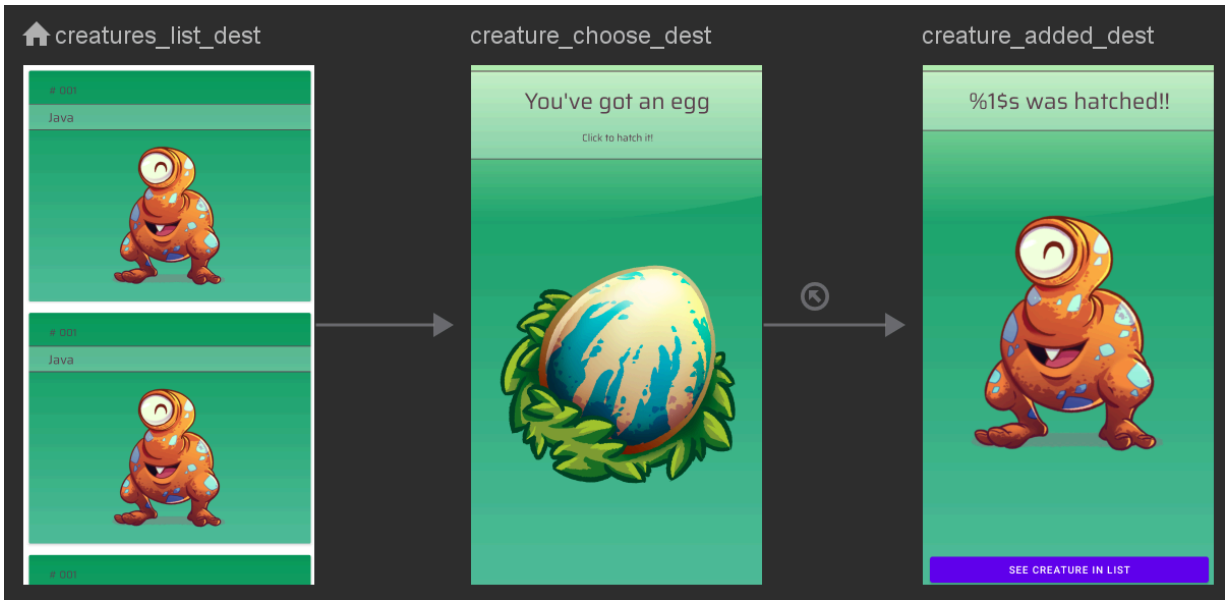


Figura 7.9: Fluxo de navegação atual.

Como a biblioteca *Navigation* lida com a gestão das *Activities* e *Fragments*, é bem simples voltar para o início da pilha de telas. Basta acessar o `NavController` e utilizar a declaração `popBackStack()`:

```
// ...
import android.widget.Button
import androidx.navigation.fragment.findNavController

@AndroidEntryPoint
class CreatureAddedFragment : Fragment() {
    // ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        // ...
        // See creature in list
        val btAddCreature = view.findViewById<Button>(R.id.btAddCreature)
        btAddCreature.setOnClickListener {
            findNavController().popBackStack()
        }
    }
}
```

Com isso, rode a aplicação e confira o resultado, como na figura a seguir. Logo de início a tela de escolha de criatura aparece; ao clicar no ovo, recebemos uma criatura aleatória, que é exibida pela tela de criatura adicionada. Ao clicar no botão `See creature in list`, a aplicação volta para a tela com a lista de criaturas.

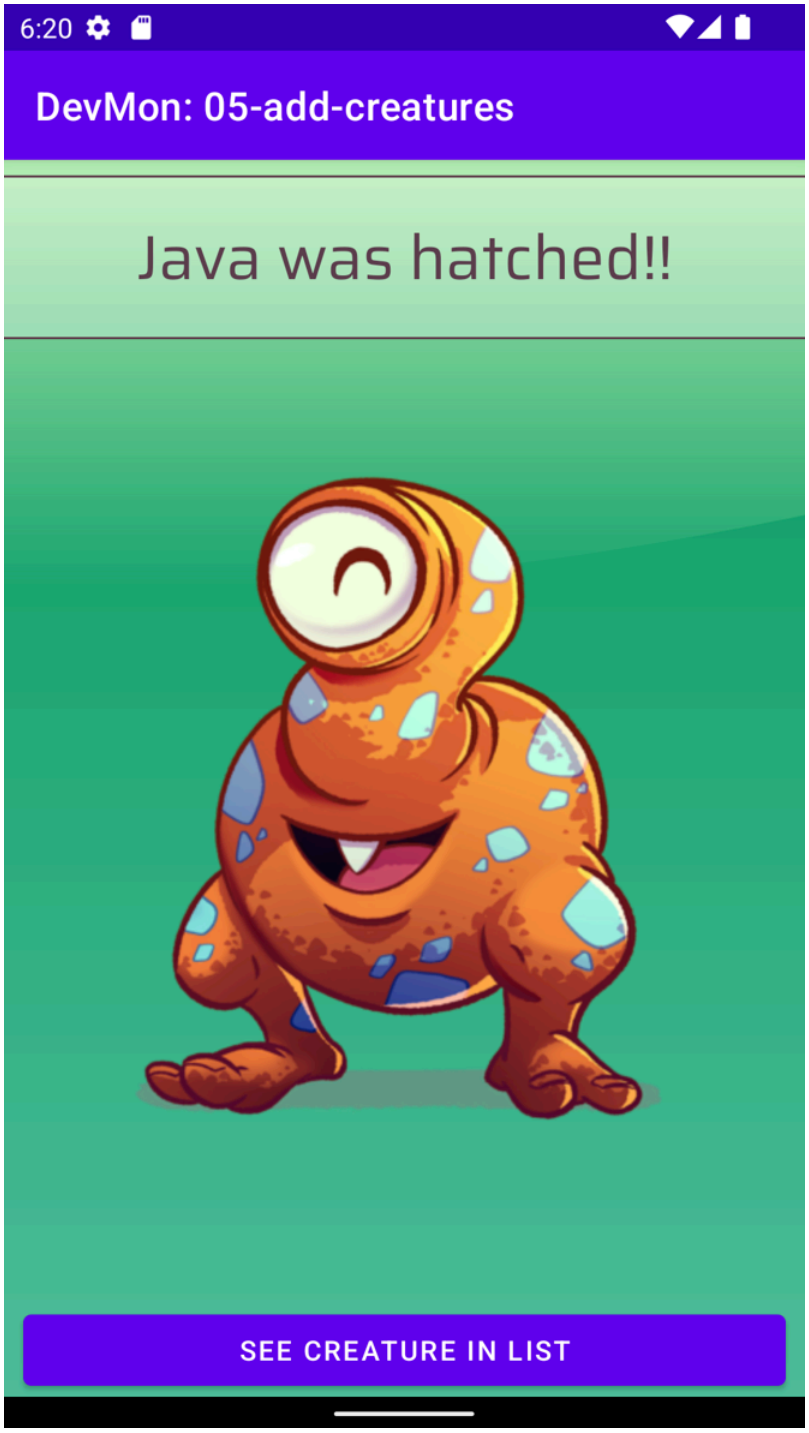


Figura 7.10: Tela de criatura adicionada.

Observação: como nem todas as criaturas foram reveladas até então, é possível receber aleatoriamente uma criatura desconhecida.

Conclusão

Com o que vimos neste capítulo, foi possível perceber que *Data Binding* e *navigation* funcionam muito bem juntos, principalmente quando o assunto é o gerenciamento de um fluxo um pouco mais complexo de telas, em que passamos informações de um local para o outro, mantendo as camadas desacopladas.

No próximo capítulo, finalizaremos a implementação da lista de criaturas, exibindo apenas a silhueta de criaturas que ainda são desconhecidas para o usuário, além de avançar em alguns outros tópicos, como a exibição de cada criatura e a adição de novas criaturas após determinado tempo.

Por último, mas não menos importante, chegou a hora de revelar que o nome do DevMon #8 é o **TypeScript**, em homenagem a essa linguagem de programação que está se demonstrando cada vez mais essencial durante o desenvolvimento de aplicações em JavaScript. O TypeScript é uma linguagem transpilada, desenvolvida pela Microsoft. Linguagens transpiladas são aquelas que transformam um código-fonte de uma linguagem (ex.: TypeScript) para outra linguagem (ex.: JavaScript), diferente das linguagens compiladas, que transformam uma linguagem (ex.: Java) em um arquivo binário (ex.: arquivo JAR).

URL da imagem: https://i.imgur.com/SUBhlhe.png

Figura 7.11: O DevMon #8 se chama TypeScript!

Capítulo 8
Silhueta das criaturas

> Caso queira visualizar as implementações completas deste capítulo, acesse o módulo `cap08-creatures-silhouette`.

No capítulo anterior, implementamos as classes de usuário e aquisição de criaturas no primeiro login da aplicação. No entanto, para que o comportamento de aquisição esteja completo, precisamos de duas funcionalidades: **esconder criaturas ainda desconhecidas** e **liberar novas criaturas após um intervalo de tempo**. Neste capítulo, vamos implementar o mecanismo de esconder as criaturas, aplicando um filtro preto por cima da imagem das criaturas que o usuário ainda não possui.

Como de costume, vamos falar sobre a criatura que será revelada neste capítulo, o DevMon #3. Nos capítulos iniciais, apresentamos o DevMon #1, Java, e o DevMon #2, Kotlin, mas ficou faltando um que é exatamente a combinação desses dois universos, mas com uma pitada de smartphone. O nome do DevMon #3 é uma homenagem ao sistema operacional mais utilizado do mundo nos anos 2017-2021. Já sabe qual é?

Figura 8.1: Qual é o DevMon #3?

Como estamos fazendo aqui no livro, exibir as silhuetas das criaturas que o usuário não possui é uma implementação bem importante para um aplicativo como o DevMon. Para construir esse comportamento na aplicação, vamos adicionar uma propriedade chamada `isKnown` na classe de *domain* `Creature.kt`. A partir disso, vamos preparar o `UserRepository` para exibir a lista completa de criaturas, mas preenchendo corretamente a propriedade `isKnown` recém-criada. Por fim, para exibir a silhueta das criaturas desconhecidas, vamos implementar um *binding adapter* chamado `enableSilhouette` que passa um filtro preto por cima da imagem da criatura.

8.1 Preparando a camada Model

Como vimos em capítulos anteriores, a camada de *domain* é responsável por estruturar toda a informação que é usada na aplicação. No caso dessa nova lógica que estamos introduzindo não é diferente. Abra o arquivo `Creature.kt` e crie uma propriedade `Boolean` chamada `isKnown` com o valor *default* `false`, para que as criaturas declaradas no arquivo `CreaturesRepository.kt` não precisem informar o valor desejado para a propriedade.

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore.model.domain

data class Creature(
    // ...
    val isKnown: Boolean = false,
)
```

Com a classe de *domain* preparada, vamos criar uma propriedade `allCreatures` dentro do arquivo `UserRepository.kt`. Sempre que algum outro código tentar obter essa informação, utilizamos a lista completa de criaturas disponível em `creaturesRepository.creatures` para combinar com a lista de criaturas que o usuário possui. Para garantir que a propriedade sempre retorne a informação mais atualizada possível, vamos utilizar a declaração `get()`. Assim, a informação é criada no momento em que alguém tenta obtê-la, da seguinte forma:

```
package
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.repository
// ...
class UserRepository @Inject constructor(
    private val creaturesRepository: CreaturesRepository,
) {
    val user = User("Paulo Salvatore", true)

    val allCreatures
        get() = creaturesRepository.creatures
```

```
        // ...
}
```

Porém, a lista com todas as criaturas não traz a informação considerando a implementação do `isKnown`, então precisamos preenchê-la. Portanto, vamos transformar uma lista de criaturas, todas com o valor de `isKnown` igual a `false`, em uma nova lista de criaturas que possui o valor de `isKnown` preenchido corretamente, sendo que o valor de `isKnown` será `true` apenas para quando a criatura estiver na lista de criaturas que o usuário possui. Para transformar uma lista em outra, podemos utilizar o operador `map`:

```
// ...
class UserRepository @Inject constructor(
    // ...
    val allCreatures
        get() = creaturesRepository.creatures.map {
            it
        }
    // ...
}
```

Note que o operador auxiliar `it` representa cada criatura existente na lista `creaturesRepository.creatures`. O próximo passo é construir uma variável que contenha o valor de `isKnown` seguindo a lógica que foi mencionada anteriormente. Para checar se a criatura em questão está na lista de criaturas do usuário, podemos utilizar o operador `any`, que retorna `true` caso o valor seja encontrado.

Como o operador `any` checa todos os itens da lista, precisamos informar uma expressão de checagem, que, no caso, será a comparação utilizando a propriedade `number` de cada criatura. Além disso, como o operador `any`, assim como o `map`, também utiliza o operador auxiliar `it`, precisamos renomeá-lo para não conflitar. Portanto, vamos renomear para `creatureOwnByUser` e utilizá-lo na checagem dos números das criaturas:

```
// ...
class UserRepository @Inject constructor(
    // ...
    val allCreatures
        get() = creaturesRepository.creatures.map {
            val isKnown = user.creatures.any { creatureOwnByUser ->
                creatureOwnByUser.number == it.number
            }

            it
        }
    // ...
}
```

Com a variável `isKnown` contendo o valor correto, vamos atualizar na criatura. No entanto, a propriedade do *domain* `Creature.isKnown` está marcada como `val`, o que impede que o valor seja alterado depois de sua criação. Manter as propriedades de *domain* como `val` é uma boa prática para impedir que alterações nesses valores sejam feitas, visto que essas alterações não refletem de forma automática no restante da aplicação. Uma técnica utilizada para preencher as propriedades `val` de `data classes` é utilizar o operador `copy`, gerado automaticamente quando adicionamos a declaração `data` para a classe:

```
// ...
class UserRepository @Inject constructor(
    // ...
    val allCreatures
        get() = creaturesRepository.creatures.map {
            // ...
            it.copy(
                isKnown = isKnown
            )
        }
    // ...
}
```

Com isso, um novo objeto de criatura é obtido, garantindo que o valor de `isKnown` está correto. Outro comportamento que podemos incluir é o de esconder o nome da criatura quando ela não for conhecida. Vamos aproveitar o `it.copy()` para alterar o nome para `?????` quando o valor de `isKnown` for `false`, utilizando uma declaração `if/else` de uma linha:

```
// ...
class UserRepository @Inject constructor(
    // ...
    val allCreatures
        get() = creaturesRepository.creatures.map {
            // ...
            it.copy(
                name = if (isKnown) it.name else "?????",
                isKnown = isKnown
            )
        }
    // ...
}
```

8.2 Preparando a camada View Model

Com a camada *Model* finalizada, vamos atualizar a camada *View Model* para pegar a lista de criaturas que acabamos de preparar a partir de `UserRepository`, em vez de `CreaturesRepository`. Vamos abrir o arquivo `CreaturesViewModel.kt` e referenciar a dependência de `UserRepository` direto no `constructor`, para garantir que o Hilt fará a injeção corretamente. A partir disso, podemos alterar a inicialização de `creatures.value`:

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore.viewmodel

// ...
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.repository.UserR
```

```
epository

@HiltViewModel
class CreaturesViewModel @Inject constructor(
    private val creaturesRepository: CreaturesRepository,
    userRepository: UserRepository,
) : ViewModel() {
    val creatures = MutableLiveData<List<Creature>>()

    init {
        creatures.value = userRepository.allCreatures
    }
    // ...
}
```

Note que a declaração `userRepository` não precisa ter a declaração `private val`, pois ela é utilizada apenas no bloco `init`, que executa automaticamente quando a classe é inicializada. No Kotlin, quando não adicionamos `val` em uma declaração dentro do construtor, estamos dizendo que ela representa um parâmetro de construtor que pode ser utilizado apenas para inicialização de propriedades. Do contrário, estamos dizendo que, além de parâmetro, ela também representa uma propriedade da classe que pode ser utilizada dentro de métodos.

É possível simplificar a declaração anterior movendo a atualização de valor de um `MutableLiveData` direto para a inicialização dele, removendo o bloco `init { }` por completo:

```
// ...
class CreaturesViewModel @Inject constructor(
    // ...
    val creatures = MutableLiveData<List<Creature>>
(userRepository.allCreatures)
    // ...
}
```

Agora que o `MutableLiveData` é autossuficiente na inicialização, ele também consegue determinar por associação o tipo da informação que está sendo aguardada. Portanto, podemos remover a declaração de tipo:

```
// ...
class CreaturesViewModel @Inject constructor(
    // ...
    val creatures = MutableLiveData(userRepository.allCreatures)
    // ...
}
```

Após remover o tipo, o `import` da classe `Creature` não está mais sendo utilizado, como é possível ver na imagem a seguir. Um atalho para otimizar os *imports* e garantir que não temos declarações que não precisam é o `Ctrl + Alt + O`.

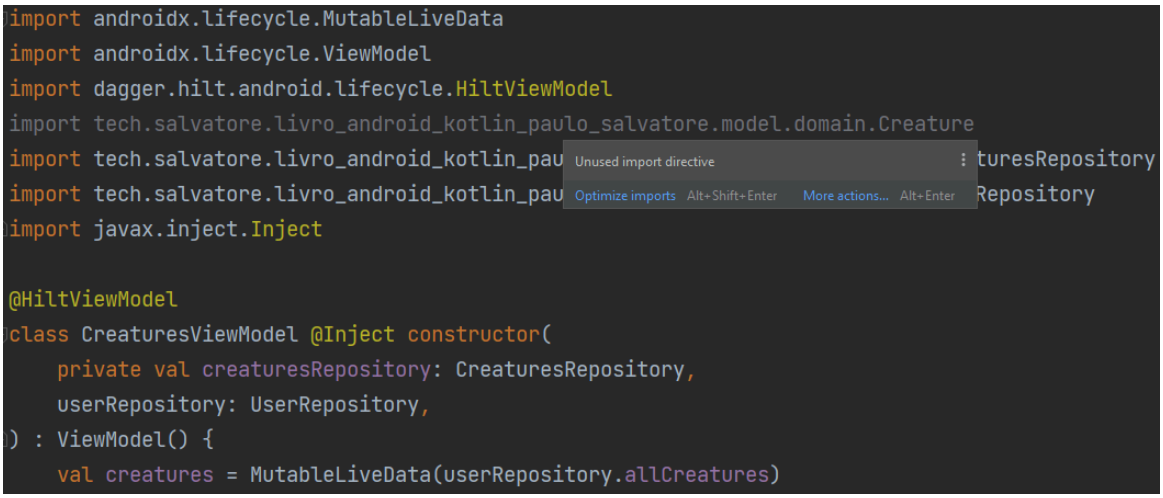


Figura 8.2: Import não utilizado

8.3 Preparando a camada View

Com tudo preparado nas camadas *Model* e *View Model*, a preparação da camada *View* se torna muito mais fácil e simples. Como neste caso não estamos apenas exibindo um texto, mas, sim, aplicando um filtro na `ImageView` que exibe a criatura, vamos começar criando o *binding adapter* que recebe um valor `boolean` para determinar se o filtro será aplicado ou não. Abra o arquivo `ImageBindingAdapter.kt`, que já contém o método de carregamento de imagens, e adicione o novo método:

```
// ...
object ImageBindingAdapter {
    // ...
    @JvmStatic
    @BindingAdapter("enableSilhouette")
    fun enableSilhouette(view: ImageView, value: Boolean) = with(view) {
    }
}
```

> Note que a declaração do Kotlin `= with(view) {` faz com que o operador `this` dentro do método represente a `ImageView`, em vez de representar a classe `ImageBindingAdapter`, facilitando a manipulação da `view` dentro de um método no *binding adapter*.

Para aplicar o filtro preto na imagem, a implementação é um *boilerplate* que não precisamos entender a fundo, mas que, em resumo, altera o `tintMode` da `ImageView` seguindo a classe `PorterDuff.Mode` do Android. Mais informações sobre a manipulação de imagens estão disponíveis em https://developer.android.com/reference/android/graphics/PorterDuff.Mode.

Figura 8.3: Documentação do Android: PorterDuff.Mode.

Após alterar o `tintMode`, alteramos também o `tintList`, que define a cor de um `ImageView`:

```
// ...
object ImageBindingAdapter {
    // ...
    @JvmStatic
    @BindingAdapter("enableSilhouette")
    fun enableSilhouette(view: ImageView, value: Boolean) = with(view) {
        ImageViewCompat.setImageTintMode(
            this,
            if (value) PorterDuff.Mode.MULTIPLY else PorterDuff.Mode.DST
        )

        if (value) {
            val black = ContextCompat.getColor(context, R.color.black)

            ImageViewCompat.setImageTintList(this,
ColorStateList.valueOf(black))
        }
    }
}
```

Após a criação do *binding adapter*, basta aplicá-lo no elemento XML da imagem da criatura, utilizando o valor de `!creature.isKnown` para determinar quando deve ativar a silhueta ou não. Abra o arquivo de *layout* `creatures_list_item.xml` , procure pela `ImageView` da criatura e adicione a declaração a seguir:

```
<ImageView
    android:id="@+id/ivCreature"
    ...
    app:enableSilhouette="@{!creature.isKnown}"
    ... />
```

Pronto! Rode a aplicação e confira o resultado. Apenas a criatura que foi obtida inicialmente está visível; todas as outras criaturas foram substituídas pela silhueta.

> Outra forma de implementar esse comportamento seria referenciando novos links de imagens, uma imagem de silhueta para cada criatura. O problema dessa implementação é que, além de precisarmos de uma imagem de silhueta para cada criatura, perdemos o recurso de cache de imagens do Glide, que é vital para reduzir o tamanho que a aplicação ocupa em disco.

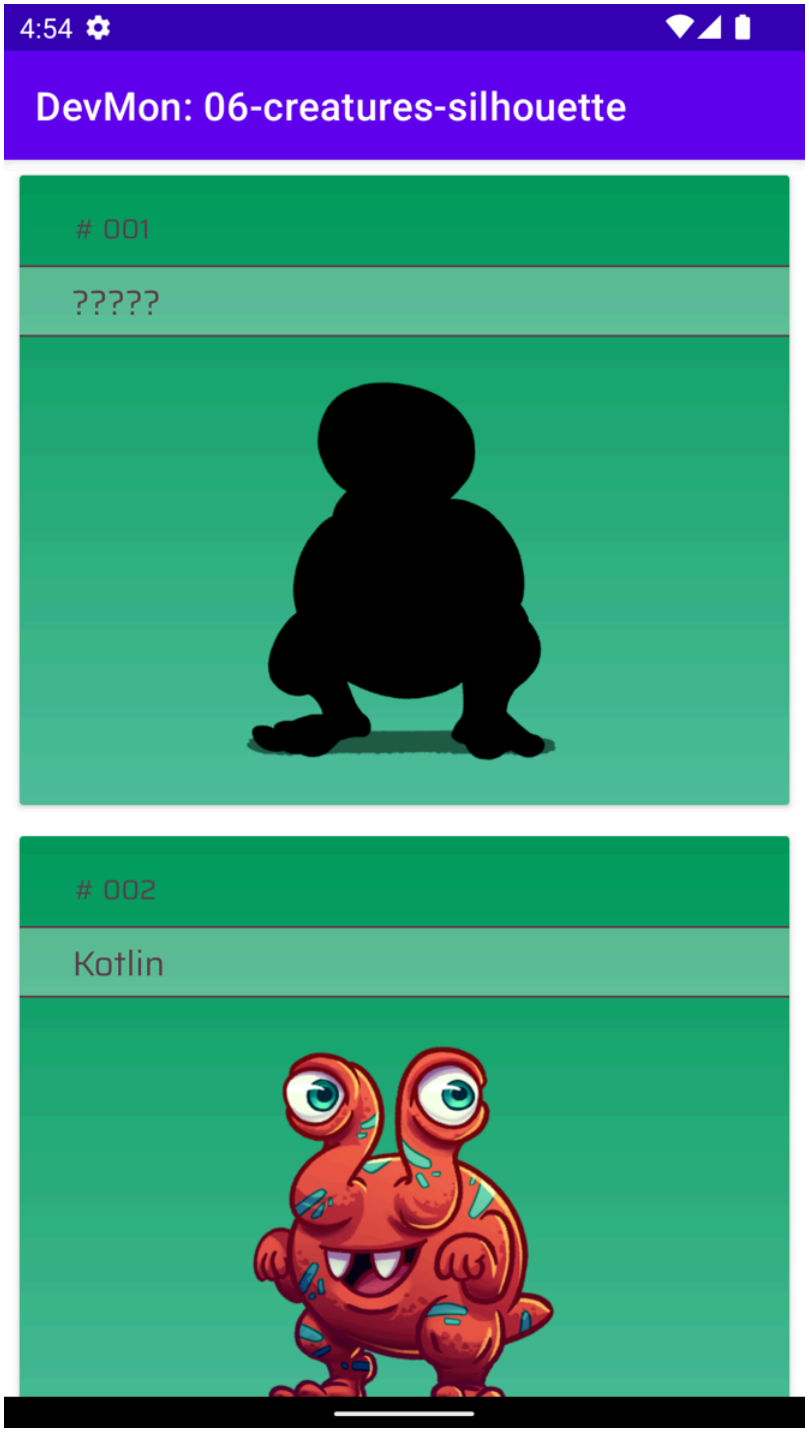


Figura 8.4: Implementação finalizada.

Conclusão

Neste capítulo, vimos que implementar novos comportamentos na arquitetura existente não é algo tão complicado e a separação das responsabilidades de cada classe é bem clara. Toda a preparação do dano fica na camada *Model*, deixando a camada *View Model* com a responsabilidade de apenas buscar a informação no lugar certo, e a camada *View* apenas implementa como aquele conteúdo deve ser apresentado.

No próximo capítulo, vamos implementar o mecanismo de adicionar novas criaturas após um determinado tempo, para que a lista de criaturas fique completa.

Por último, mas não menos importante, chegou a hora de revelar que o nome do DevMon #3 é **Android**! Como mencionado, o Android é o sistema operacional mais utilizado no mundo, superando o Windows em 2016 graças ao poder dos smartphones. Outro motivo que fez esse sistema operacional baseado no núcleo Linux ter ganhado tanta popularidade é o fato dele ser *open source*, o que permite com que as empresas façam as próprias modificações e comercializem dispositivos com essas versões customizadas. Escrito na linguagem C, como quase todos os outros sistemas operacionais, possui duas linguagens oficiais: Kotlin e Java. Porém, as aplicações contêm apenas código Java, visto que o Kotlin é transpirado durante a compilação.

URL da imagem: https://i.imgur.com/r4UjONq.png

Figura 8.5: O DevMon #3 se chama Android!

CAPÍTULO 9
Visualizando criaturas

> Caso queira visualizar as implementações completas deste capítulo, acesse o módulo `CAP09-CREATURES-VIEW`.

Poder visualizar um pouco mais de detalhes das criaturas que acabaram de ser obtidas é uma funcionalidade indispensável em uma aplicação como o DevMon. Para implementar essa nova funcionalidade, precisamos criar um `Fragment` e inseri-lo no fluxo de navegação, além de adaptar a lista para que, ao clicar em cada item, seja feito o direcionamento para esse novo `Fragment`. Por fim, é necessário fornecer os dados individuais das criaturas através de um novo `ViewModel` dedicado apenas para a tela de visualização da criatura.

Antes da implementação, como de costume, vamos falar sobre a criatura que será revelada neste capítulo, o DevMon #9. Desta vez, o nome da criatura é em homenagem a um dos frameworks de back-end em Node.JS/JavaScript mais conhecidos do cenário. Uma de suas principais características é a sua leveza, o que permite a criação de uma aplicação back-end com apenas algumas poucas linhas de código. O back-end do DevMon, inclusive, é desenvolvido tendo este framework como o motor da aplicação. E aí, já sabe qual é?

Figura 9.1: Qual é o DevMon #9?

Para facilitar o entendimento da implementação que faremos neste capítulo, vamos seguir este fluxo:

1. Assim que o aplicativo abre, a lista de criaturas é carregada;
2. O usuário clica na criatura desejada;
3. Pegamos o número da criatura clicada e abrimos uma tela de visualização de criatura;
4. Assim que a tela abre, carregamos um `ViewModel` e enviamos o número da criatura;
5. O `ViewModel` busca a informação no *Repository*;
6. Assim que a informação estiver pronta, ela é disponibilizada em uma variável `LiveData` para que seja observada pela camada

View, através do *Data Binding* no arquivo XML.

9.1 Preparando a camada View Model

Vamos preparar a camada *View Model*, pois precisamos buscar o dado na camada *Repository* através dela. A classe que vamos criar precisa ser capaz de realizar duas tarefas: a primeira é receber um número de criatura e buscar essa informação na classe do *Repository*; a segunda é disponibilizar a criatura carregada em uma variável `LiveData`, para que a camada *View* consiga observar mudanças nessa variável.

Para começar, vamos criar um arquivo chamado `CreatureViewModel` dentro do pacote `viewmodel`. Dentro desse arquivo, certifique-se de preparar a estrutura inicial de um *View Model*, deixando-o visível para o Hilt com a *annotation* `@HiltViewModel` na classe e inserindo o `@Inject` para o construtor primário. Além disso, para que a classe seja um `ViewModel` válido do Android, precisamos estender à classe `ViewModel` do pacote `androidx.lifecycle`, da seguinte forma:

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore.viewmodel

import androidx.lifecycle.ViewModel
import dagger.hilt.android.lifecycle.HiltViewModel
import javax.inject.Inject

@HiltViewModel
class CreatureViewModel @Inject constructor() : ViewModel() {
}
```

Agora que a estrutura inicial do `CreatureViewModel` está preparada, vamos implementar o primeiro mecanismo, o de receber um número de criatura e carregar a informação na camada *Repository*. Para isso, precisamos que o Hilt disponibilize a classe `CreaturesRepository`;

portanto, a dependência deve ser declarada diretamente no construtor, como no trecho a seguir.

```
// ...
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.repository.Creat
uresRepository
// ...
class CreatureViewModel @Inject constructor(
    private val creaturesRepository: CreaturesRepository,
) : ViewModel() {
    // ...
```

Com a dependência devidamente injetada pelo Hilt, podemos utilizá-la em qualquer método da classe.

Vamos preparar o novo método que recebe o número da criatura, carrega a criatura na camada *Repository* e envia para o `LiveData`. Assim como fizemos na implementação do `UserViewModel` no capítulo *Adquirindo criaturas*, vamos deixar o `MutableLiveData` apenas para uso interno, expondo apenas o `LiveData` para que não seja possível alterar seu valor estando fora da classe `CreatureViewModel`.

```
// ...
import androidx.lifecycle.MutableLiveData
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.domain.Creature

@HiltViewModel
class CreatureViewModel @Inject constructor(
    // ...
    private val _creature = MutableLiveData<Creature>()

    val creature: LiveData<Creature>
        get() = _creature

    fun loadCreature(number: Int) {
        _creature.value = creaturesRepository.findCreature(number)
```

```
    }
}
```

9.2 Criando a tela de visualização da criatura

Com o `CreatureViewModel` pronto, vamos criar o `Fragment` que exibe a criatura que acabou de ser clicada. Utilizando como referência a preparação de `Fragments` dos capítulos anteriores, vamos construir o `CreatureViewFragment.kt` seguindo estes passos:

1. Crie a classe `CreatureViewFragment.kt` no pacote `view.creatures`;
2. Crie o arquivo de *layout* `creature_view_fragment.xml` com o *Data Binding* ativado, na pasta `res/layout`;
3. Importe o *binding* gerado automaticamente pelo Android dentro do `Fragment`. Como o nome do *binding* gerado é baseado no nome do arquivo de *layout*, nesse caso, será o `CreatureViewFragmentBinding`;
4. Importe o `ViewModel` criado anteriormente com a declaração `by viewModels()`;
5. Adicione o código *boilerplate* para obtenção da instância do `binding`, sobrescrevendo o método `onCreateView()`;
6. Assim que a *View* estiver criada, é possível obter o número da criatura utilizado para abri-la e enviar para o *View Model* carregar a criatura em questão;
7. Para exibir as informações da criatura carregada direto no *layout* utilizando o *Data Binding*, é necessário passar a referência do *View Model*.

Para começar, crie a classe `CreatureViewFragment.kt` no pacote `view.creatures` com o seguinte código *boilerplate*:

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore.view.creatures

import android.os.Bundle
import android.view.LayoutInflater
```

```
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment
import androidx.fragment.app.viewModels
import dagger.hilt.android.AndroidEntryPoint
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.databinding.CreatureVi
ewFragmentBinding
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.viewmodel.CreatureView
Model

@AndroidEntryPoint
class CreatureViewFragment : Fragment() {
    private lateinit var binding: CreatureViewFragmentBinding

    private val viewModel: CreatureViewModel by viewModels()

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?,
    ): View {
        binding =
            CreatureViewFragmentBinding.inflate(
                layoutInflater,
                container,
                false
            )

        return binding.root
    }
}
```

Para que o *Data Binding* funcione corretamente, é necessário criar o arquivo de *layout*. Antes de criar a tela, vamos dar uma olhada na imagem a seguir para conhecer como deve ser o seu *layout*.

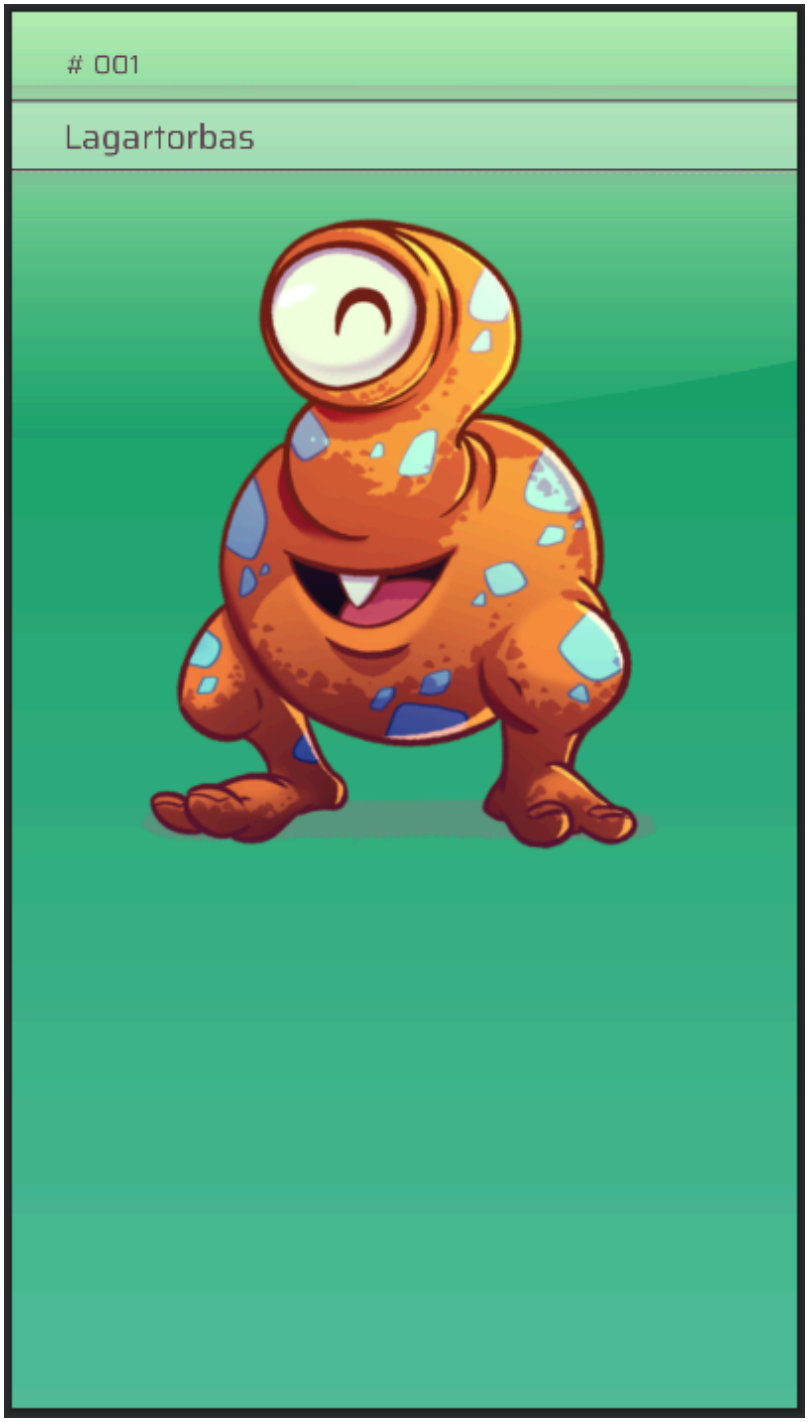


Figura 9.2: Layout da tela de visualizar criaturas.

Dentro da pasta `res/layout`, crie um arquivo chamado `creature_view_fragment.xml` e adicione o conteúdo *boilerplate* a seguir.

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">

    <data>

        <variable
            name="viewModel"

type="tech.salvatore.livro_android_kotlin_paulo_salvatore.viewmodel.CreatureViewModel" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:id="@+id/main"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@drawable/screen_background"
        tools:context=".view.creatures.list.CreaturesViewFragment">

        <ImageView
            android:id="@+id/ivBackgroundTop"
            android:layout_width="0dp"
            android:layout_height="221dp"
            android:scaleType="fitXY"
            android:src="@drawable/screen_top_background"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            tools:ignore="ContentDescription" />

    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Para implementar o *layout* proposto, vamos utilizar como referência o *layout* dos itens da lista, criados no capítulo *Lista de criaturas*, seguindo este fluxo:

1. Número da criatura com a formatação correta;
2. Nome da criatura com o background;
3. Imagem da criatura.

Dentro do elemento `<ConstraintLayout>`, adicione os elementos a seguir, preenchendo os valores com as informações disponíveis dentro da propriedade `creature` no `CreatureViewModel`.

```
<?xml version="1.0" encoding="utf-8"?>
<layout ...>
    <androidx.constraintlayout.widget.ConstraintLayout ...>

        <ImageView ...>

        <TextView
            android:id="@+id/tvCreatureNumber"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginStart="28dp"
            android:layout_marginLeft="28dp"
            android:layout_marginTop="16dp"
            android:fontFamily="@font/saira_400"

android:text="@{@string/creature_number(viewModel.creature.number)}"
            android:textColor="#5D414F"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            tools:text="# 001" />

        <androidx.constraintlayout.widget.ConstraintLayout
            android:id="@+id/constraintLayoutTitleSection"
            style="@style/creatures_background_title_section"
            android:layout_height="37dp"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@id/tvCreatureNumber">

            <TextView
                android:id="@+id/tvCreatureName"
```

```
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_marginStart="27dp"
                android:layout_marginLeft="27dp"
                android:fontFamily="@font/saira_400"
                android:text="@{viewModel.creature.name}"
                android:textColor="#5D414F"
                android:textSize="18sp"
                app:layout_constraintBottom_toBottomOf="parent"
                app:layout_constraintStart_toStartOf="parent"
                app:layout_constraintTop_toTopOf="parent"
                tools:text="Creature Name" />

        </androidx.constraintlayout.widget.ConstraintLayout>

        <ImageView
            android:id="@+id/ivCreature"
            android:layout_width="340dp"
            android:layout_height="340dp"
            android:layout_marginTop="20dp"
            android:contentDescription="@string/creatures_image"
            app:imageUrl="@{viewModel.creature.imageUrl}"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"

app:layout_constraintTop_toBottomOf="@id/constraintLayoutTitleSection"
            tools:src="@drawable/creature_java" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Com isso, o arquivo de *layout* está pronto para receber a instância do `ViewModel`. Entretanto, o `Fragment` precisa carregar essa informação para que ela possa ser inserida. Para que a tela de visualização de criatura possa ser aberta, é necessário que ela esteja referenciada no fluxo de navegação. Vamos abrir o arquivo `nav_graph.xml` e adicionar o novo destino com o trecho a seguir.

```
<?xml version="1.0" encoding="utf-8"?>
<navigation ...>
```

```
    <!-- ... -->
    <fragment
        android:id="@+id/creature_view_dest"

android:name="tech.salvatore.livro_android_kotlin_paulo_salvatore.view.cre
atures.CreatureViewFragment"
        android:label="CreatureViewFragment"
        tools:layout="@layout/creature_view_fragment">
    </fragment>
</navigation>
```

Como é necessário saber qual a criatura que a tela exibirá, precisamos adicionar um argumento chamado `creatureNumber`, do tipo `integer`, da seguinte forma.

```
<?xml version="1.0" encoding="utf-8"?>
<navigation ...>
    <!-- ... -->
    <fragment android:id="@+id/creature_view_dest" ...>
        <argument
            android:name="creatureNumber"
            app:argType="integer" />
    </fragment>
</navigation>
```

Com isso, podemos acessar o argumento que é passado quando a tela é criada e utilizá-lo para carregar a criatura. Dentro do arquivo `CreatureViewFragment`, vamos sobrescrever o método `onViewCreated()`, obter os argumentos através da declaração `by navArgs()`, extrair o `creatureNumber` e executar o método `viewModel.loadCreature()` para que a criatura seja carregada no mesmo instante em que a tela abre.

```
// ...
import androidx.navigation.fragment.navArgs

@AndroidEntryPoint
class CreatureViewFragment : Fragment() {
    // ...
```

```
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        val safeArgs: CreatureViewFragmentArgs by navArgs()
        val creatureNumber = safeArgs.creatureNumber

        viewModel.loadCreature(creatureNumber)

        binding.viewModel = viewModel
    }
}
```

9.3 Abrir a tela ao clicar no item da lista

Agora que a tela de visualização da criatura está pronta, precisamos abri-la assim que o card da criatura na lista for clicado. Para isso, vamos atualizar a classe `CreaturesListAdapter` para permitir que comportamentos customizados sejam executados.

Um jeito bem interessante de realizar essa implementação é receber um parâmetro do construtor que seja uma função. Dessa forma, o `Adapter` não precisa saber qual comportamento será executado, podendo receber qualquer comportamento customizado e apenas executá-lo. Neste caso, como estamos executando o comportamento para cada item da lista, é interessante que essa função saiba qual é a criatura que está sendo exibida naquele item específico. Para isso, podemos utilizar a declaração `(Creature) -> Unit`, que indica que a função passada receberá um primeiro parâmetro do tipo `Creature` e não terá retorno algum, portanto, utilizamos o tipo `Unit` do Kotlin, que é equivalente ao `void` do Java. Para concluir a implementação, vamos executar a função assim que o `itemView.setOnClickListener` disparar. Vamos ao código:

```
// ...
class CreaturesListAdapter(
```

```
    // ...
    private val listener: (Creature) -> Unit,
) /* ... */ {
    inner class ViewHolder(/* ... */) {
        fun bindView(
            // ...
            listener: (Creature) -> Unit,
        ) {
            // ...
            itemView.setOnClickListener { listener(item) }
        }
    }
    // ...
    override fun onBindViewHolder(/* ... */) {
        // ...
        holder.bindView(item, listener)
    }
    // ...
}
```

Ao realizar essa modificação, a classe `CreaturesListFragment` começa a apresentar um erro, pois o parâmetro `listener` que acabamos de declarar como obrigatório não está sendo informado. Navegue até o método `onViewCreated` e busque pela declaração do *adapter* `recyclerView.adapter = CreaturesListAdapter(it)`.

Como o parâmetro `listener` da classe `CreatureListAdapter` é uma função e está posicionado na última posição, podemos utilizar a expressão `lambda` para implementá-lo:

```
// ...
@AndroidEntryPoint
class CreaturesListFragment : Fragment() {
    // ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        // ...
        creaturesViewModel.creatures.observe(viewLifecycleOwner) {
            recyclerView.adapter = CreaturesListAdapter(it) { creature ->
            }
```

```
        }
    }
}
```

Para que a tela de visualização da criatura abra no momento que o item da lista for clicado, precisamos criar uma `action` no arquivo `nav_graph.xml`, como foi feito anteriormente no capítulo *Adquirindo criaturas*, utilizando a interface gráfica do arquivo de navegação. Abra o arquivo mencionado e crie uma conexão entre as telas `CreaturesListDest` e `CreatureViewDest`, como mostrado na imagem a seguir.

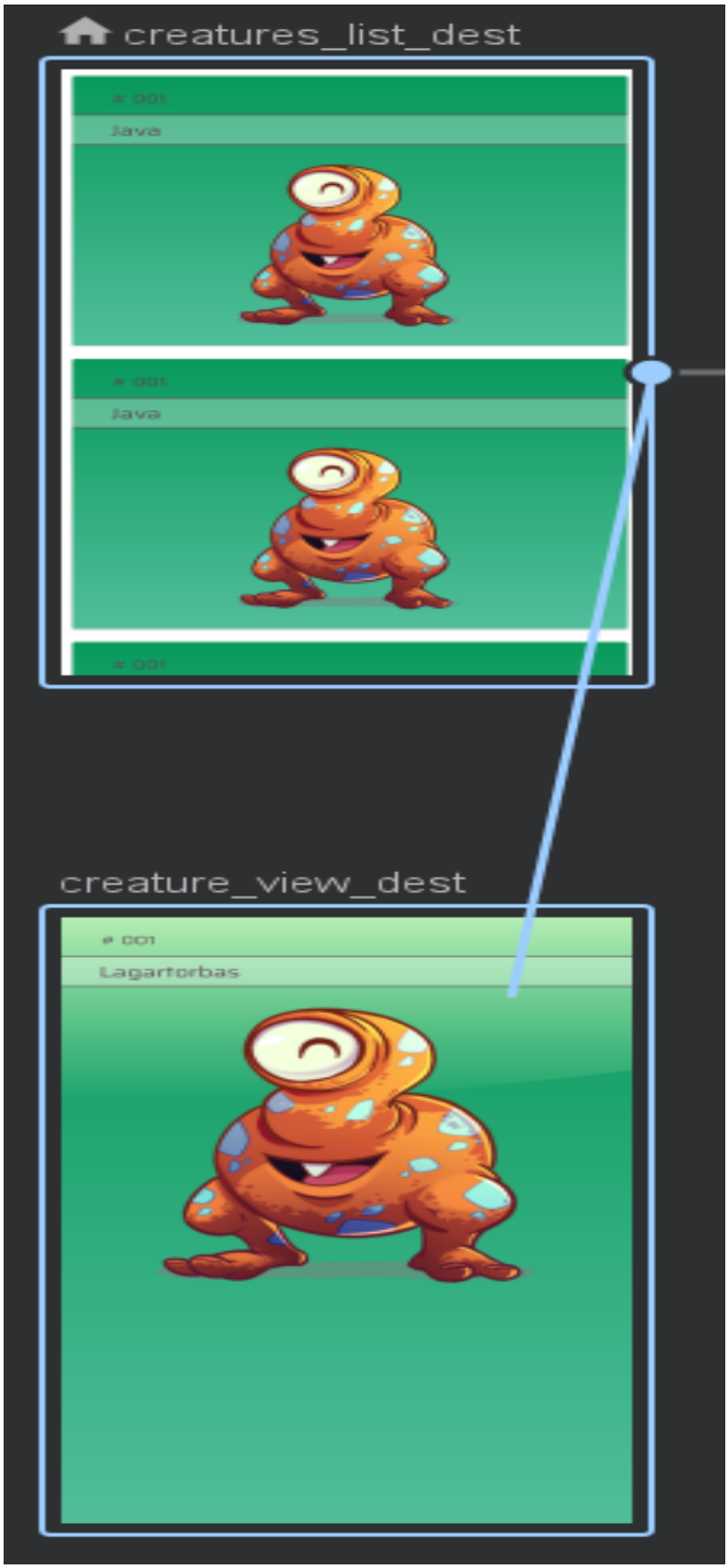

creatures_list_dest
001
Java
001
Java
001
creature_view_dest
001
Lagartorbas

Figura 9.3: Conexão entre as telas de lista e visualização.

Com a conexão criada, altere o ID da `action` para `@+id/creature_view_action`. O código do `nav_graph.xml` deve ficar como o trecho a seguir.

```
<?xml version="1.0" encoding="utf-8"?>
<navigation ...>
    <fragment android:id="@+id/creatures_list_dest" ...>
        <!-- ... -->
        <action
            android:id="@+id/creature_view_action"
            app:destination="@id/creature_view_dest" />
    </fragment>
</navigation>
```

Para que a tela de visualização da criatura seja aberta, é necessário obter a instância da `action`, informando o `number` da criatura e executando o método `navigate()` do `NavController`, assim como fizemos nos capítulos anteriores. Portanto, dentro da implementação do `listener` do `CreatureListAdapter`, adicionamos o seguinte trecho.

```
// ...
class CreaturesListFragment : Fragment() {
    // ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        // ...
        creaturesViewModel.creatures.observe(viewLifecycleOwner) {
            recyclerView.adapter = CreaturesListAdapter(it) { creature ->
                val action =
                    CreaturesListFragmentDirections
                        .creatureViewAction(creature.number)

                findNavController().navigate(action)
            }
        }
    }
}
```

Para testar, vamos rodar a aplicação e clicar em alguma criatura da lista para visualizá-la. O resultado será como o da imagem a seguir.

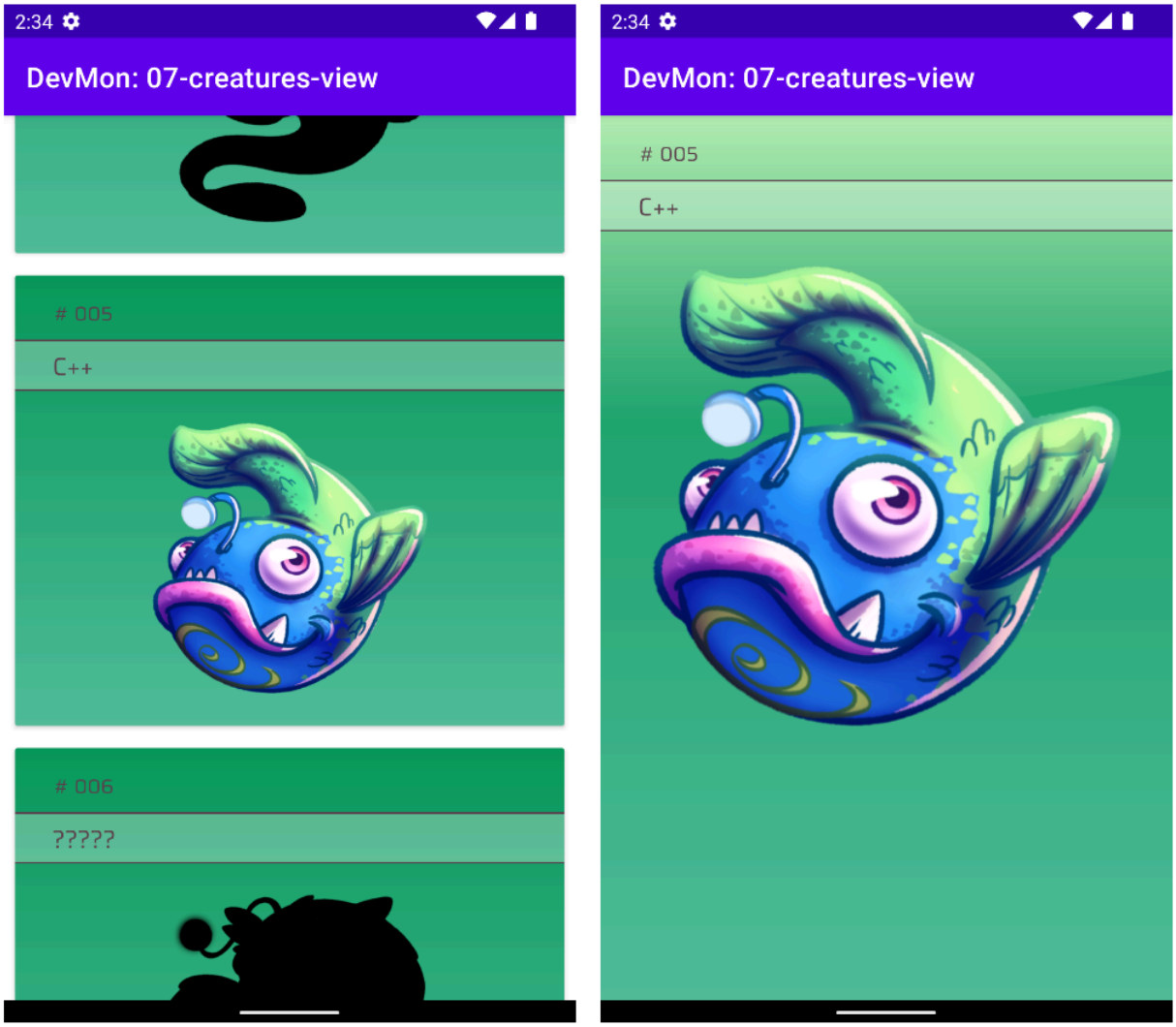


Figura 9.4: Visualizando criaturas.

Note que, ao clicar em criaturas desconhecidas, a tela de visualização aparece e exibe a criatura que, até então, estava escondida. Para corrigir, é necessário verificar se a propriedade `isKnown` da criatura em questão está com o valor `false`, encerrando a execução do `listener` com um `return` vazio. Como a declaração está dentro do método `onViewCreated`, é necessário especificar que o `return` deve ser aplicado ao `CreaturesListAdapter`, da seguinte forma.

```
// ...
class CreaturesListFragment : Fragment() {
    // ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        // ...
        creaturesViewModel.creatures.observe(viewLifecycleOwner) {
            recyclerView.adapter = CreaturesListAdapter(it) { creature ->
                if (!creature.isKnown) {
                    return@CreaturesListAdapter
                }
                // ...
            }
        }
    }
}
```

Com isso, apenas criaturas conhecidas poderão receber eventos de clique, o que conclui a nossa implementação.

Conclusão

Neste capítulo, construímos a visualização de criaturas: criamos a tela, inserimo-la no fluxo de navegação do app e implementamos o mecanismo de clique na lista de criaturas.

No próximo capítulo, vamos integrar a aplicação a uma API exclusiva do projeto, o que permitirá atualizar a lista de criaturas pela internet sem que uma nova build seja necessária. Para trabalhar as requisições para internet, vamos utilizar a biblioteca RX, entendendo como ela facilita o trabalho com chamadas assíncronas e torna a aplicação mais robusta e flexível.

Por último, mas não menos importante, chegou a hora de revelar que o nome do DevMon #9 é o **Express**, inspirado no framework de back-end para Node.JS que, apesar de simples, consegue ser bem poderoso e ter uma performance muito semelhante a outros frameworks mais complexos. Se estiver pensando em se aventurar no universo back-end, o Express pode ser um início bem fácil e

trazer diversos conceitos importantes que devem te ajudar no universo das APIs.

URL da imagem: https://i.imgur.com/pRGNm6T.png

Figura 9.5: O DevMon #9 se chama Express!

CAPÍTULO 10
ReactiveX (RX)

> Caso queira visualizar as implementações completas deste capítulo, acesse o módulo `CAP10-RX`.

O ReactiveX (RX, a partir de agora) é uma biblioteca de código aberto, desenvolvida pela Netflix, muito poderosa e disponível em diversas linguagens de programação, como Java, Kotlin, C++, JavaScript, C#, PHP, e até em alguns softwares e frameworks, como Android, .NET, Unity, entre outros. Construída utilizando o conceito da Programação Reativa (falaremos mais a seguir), tem como objetivo facilitar o desenvolvimento das aplicações, fornecendo um kit de funções bem poderoso para manipular informações em geral, independente de estarem em um banco de dados local ou na web.

Antes da implementação, como de costume, vamos falar sobre a criatura que será revelada neste capítulo, o DevMon #10. O nome da criatura é em homenagem a um outro framework de back-end em Node.JS/JavaScript/TypeScript que vem fazendo a diferença no cenário. Entre suas principais características estão sua arquitetura e a facilidade/simplicidade em resolver problemas comuns. Com isso, permite a criação de uma aplicação back-end muito robusta sem precisar desenvolver soluções complexas, fornecendo um alto nível de abstração. O back-end do DevMon usa esse framework para abstrair as implementações com o Express (apresentado no capítulo anterior). E aí, já sabe qual é?

Figura 10.1: Qual é o DevMon #10?

10.1 Programação Reativa

Para começar a falar sobre o assunto, precisamos distinguir dois conceitos: o de Programação Reativa e o de RX. **Programação Reativa** (*Reactive Programming*) é um paradigma da programação que diz que devemos nos preocupar com o fluxo de dados e com as mudanças que esse fluxo sofre ao longo do percurso. O **RX**, por sua vez, é uma biblioteca de código aberto que busca facilitar a implementação do paradigma, trazendo ferramentas preparadas

para trabalhar com os fluxos de dados, além de facilitar o uso da programação assíncrona e a comunicação de diferentes partes do código.

Neste capítulo, entenderemos algumas questões iniciais sobre o manifesto reativo para entender os objetivos principais da Programação Reativa, além de explorar um pouco mais o RX para entender como essa biblioteca está estruturada e o que exatamente podemos fazer com ela.

Pilares da Programação Reativa

De acordo com o manifesto reativo, publicado em 2014, a Programação Reativa possui 4 pilares principais: **responsividade**, **resiliência**, **elasticidade** e **orientação a mensagens**, conforme é possível visualizar na figura a seguir.

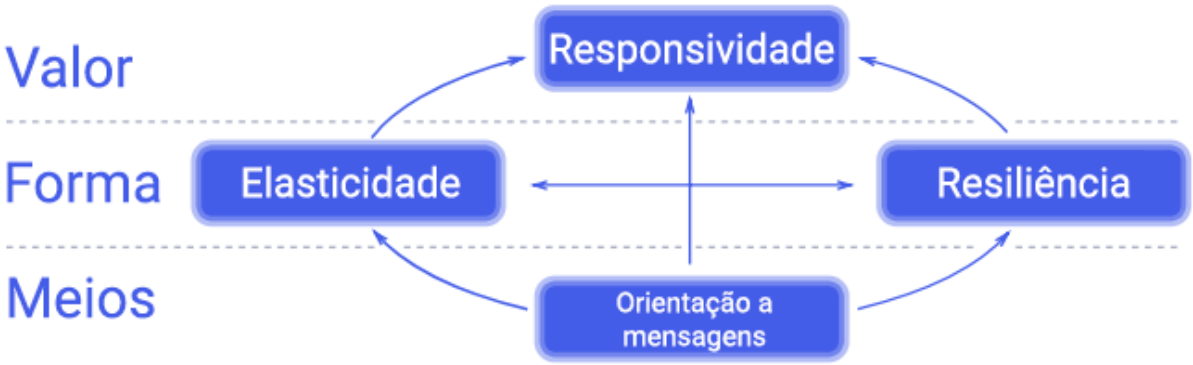


Figura 10.2: Os 4 pilares da Programação Reativa.

1. **Responsividade:** a aplicação deve responder o mais rápido possível e problemas devem ser detectados e resolvidos com a máxima eficácia;
2. **Resiliência:** a aplicação deve continuar respondendo em caso de falha;
3. **Elasticidade:** a aplicação deve continuar funcionando de forma plena mesmo com aumento da demanda;

4. **Orientação a mensagens:** em vez de interligar diretamente as partes de um software, tente pensar como pequenos ecossistemas isolados que disparam mensagens claras sobre o que está acontecendo em seu interior. A partir disso, outros pedaços do software saberão entendê-lo de uma forma mais clara.

Como é possível ver na figura, todos os pilares estão relacionados de alguma maneira e, assim, ao priorizar a performance e facilitar a manutenção, conseguimos garantir que tanto o código quanto a aplicação atinjam um alto nível de qualidade.

10.2 Fluxo de dados síncrono vs. assíncrono

Para conseguir entender mais sobre o RX, precisamos de uma boa definição do que é o fluxo de dados e qual a diferença entre esse fluxo ser síncrono ou assíncrono.

O **fluxo de dados** é o nome dado ao processo de trabalhar com uma informação que passa por algumas etapas antes de estar pronta para aparecer para o usuário. Na nossa aplicação, por exemplo, para exibir uma lista com todas os DevMon que conhecemos, precisamos executar uma chamada para um back-end, que traz uma `string` com todo o dado. A partir disso, essa `string` é transformada em uma informação que a linguagem de programação é capaz de manipular com mais facilidade.

Em alguns casos, se a informação não vier completa na primeira requisição, pode ser necessário realizar novas requisições e processar todos esses resultados de uma só vez, antes de deixar tudo pronto para a exibição na tela. Toda essa manipulação da informação, tanto a obtenção quanto a transformação, é chamada de **fluxo de dados**.

O **fluxo de dados síncrono** é aquele em que uma etapa só acontece depois que a etapa anterior foi finalizada. Na nossa aplicação, por exemplo, as imagens dos DevMon só serão carregadas assim que a lista de DevMon for recebida. Além disso, se o carregamento das imagens for feito da maneira síncrona, a imagem do segundo DevMon só será carregada após o carregamento da imagem do primeiro ser concluído.

O **fluxo de dados assíncrono** é aquele em que uma etapa acontece ao mesmo tempo que outra. Na nossa aplicação, as imagens de todos os DevMon são carregadas ao mesmo tempo, disparando várias requisições em paralelo. Nesse caso, apesar de ser necessário aguardar a primeira requisição para saber qual a lista de imagens, o carregamento das imagens em si fica muito mais rápido, pois não precisamos aguardar cada imagem terminar de carregar para disparar o carregamento da próxima, visto que uma não depende da outra.

Agora que entendemos o que é o fluxo de dados e a diferença entre fluxos síncronos e assíncronos, fica um pouco mais claro entender como toda aplicação gira em torno de fluxos de dados tanto síncronos quanto assíncronos. Na aplicação que estamos construindo, os dados dos DevMon e dos usuários estão armazenados em variáveis na memória e estão sendo acessados de maneira **síncrona**. No entanto, as imagens do DevMon estão utilizando endereços da internet, que são carregados pela biblioteca Glide, que carrega várias imagens ao mesmo tempo, são processadas de maneira **assíncrona**.

No próximo capítulo, vamos integrar o back-end à aplicação Android. Para isso, vamos precisar implementar requisições para servidores na web, o que deve levar alguns segundos para concluir. Quando precisamos fazer com que a aplicação aguarde um certo tempo para que algo seja concluído, é necessário utilizar a programação assíncrona, pois, caso contrário, travaríamos o app aguardando o resultado, o que tem impacto direto na experiência de usuário.

Como implementar chamadas assíncronas?

Quando executamos um programa em um computador ou em um celular, estamos executando um **processo** direto na CPU. Esses processos possuem ***threads***, que funcionam como pequenos ecossistemas capazes de executar declarações em sequência. Para implementar chamadas que acontecem ao mesmo tempo, ou seja, chamadas assíncronas, criamos *threads* que executam as instruções que foram passadas e comunicam apenas o resultado daquela instrução.

Para conseguir coordenar todo esse processamento, geralmente temos uma *thread* inicial, comumente chamada de *main thread*, que é responsável por iniciar todas as outras *threads*, conforme a necessidade. No Android, ela também é conhecida como *UI Thread*, pois, além de ser a *thread* principal, é responsável pela renderização da interface de usuário (UI).

Existem várias formas de implementar chamadas assíncronas no Android, como: `Thread/Runnable`, `AsyncTask`, `Coroutines` etc. No entanto, gerenciar a comunicação entre *threads* nem sempre é uma tarefa fácil, já que exige muito planejamento e muita implementação de código *boilerplate*, principalmente quando queremos realizar operações mais complexas.

Foi justamente para facilitar o trabalho com fluxos de dados síncronos e assíncronos, além da comunicação entre esses fluxos, que surgiu a biblioteca RX. Ela fornece uma forma relativamente simples de implementação, a partir do momento em que entendemos alguns princípios básicos.

10.3 A base do RX

Agora que entendemos o problema que o RX resolve, vamos entender o funcionamento de quatro componentes principais: `Observables`, `Observers`, `Diposables` e `Operators`.

Observable

O primeiro item da nossa lista é o `Observable`, onde tudo começa. É nele que definimos o que será realizado, a base da nossa operação. Podemos definir um `Observable` como algo que está emitindo eventos para algum fluxo de dados, podendo ser qualquer coisa sobre a qual estamos interessados em saber mais. A partir da criação do `Observable`, temos um fluxo de dados **observável**, ou seja, quem tiver interesse nas informações, pode observar e receber eventos.

Neste exemplo de código, temos um `Observable` que emite um evento para cada um dos números informados:

```
Observable.just(1, 2, 3)
```

Neste outro exemplo de código, que usa a biblioteca *RxBinding*, temos um `Observable` que emite um evento sempre que o texto de um `EditText` é alterado:

```
val nameChangeObservable = RxTextView.textChanges(etName)
```

> Um fluxo de dados do RX sempre começa a partir de um `Observable`.

Observers

No entanto, um `Observable` sozinho não faz muita coisa, visto que não basta ter os eventos, precisamos fazer algo com eles, certo? Para isso, temos os `Observers`, que observam todas as novas informações que o `Observable` está deixando disponíveis. Até então, para `Observables` que possuem uma quantidade fixa de evento a ser emitidas, não precisamos de um `dispose()`:

```
val observable = Observable.just(1, 2, 3)

observable.subscribe {
  println(it * 2) // Irá exibir 2, 4 e 6
}
```

Se olharmos para a declaração da interface em Java, podemos observar exatamente o seguinte:

```
public interface Observer<T> {
   void onSubscribe(@NonNull Disposable d);
   void onNext(@NonNull T t);
   void onError(@NonNull Throwable e);
   void onComplete();
}
```

Inicialmente, a interface recebe um tipo `T`, que significa o tipo de variável que o `Observer` vai observar. Depois, temos o método `onSubscribe()`, que é chamado assim que o `Observable` começa a ser observado. Um objeto da classe `Disposable` é trazido quando chamamos esse método. Entenderemos o papel o `Disposable` mais à frente.

O principal método é o `onNext()`, que recebe cada item informado do tipo `T`, e é onde utilizaremos o valor já processado. O `onError()` atua como um centralizador de qualquer erro ocasionado durante a operação realizada, onde podemos declarar a lógica para tratar esses erros recebidos.

Por último, temos o `onComplete()`, que é chamado assim que toda a operação termina. Ele só é disparado caso a operação tenha sido um sucesso.

`Consumer`: um `Observer` simplificado

Além do `Observer`, temos a interface `Consumer`, que facilita uma implementação do `subscribe`. Nessa interface, implementamos a passagem no método `onNext()`, que, nesse caso, é chamado de

`accept()`. O conteúdo da interface é bem simples, como pode ser observado em sua declaração, em Java.

```
public interface Consumer<T> {
   /**
    * Consume the given value.
    * @param t the value
    * @throws Exception on error
    */
   void accept(T t) throws Exception;
}
```

Em alguns casos, quando não é necessária a implementação dos quatro métodos de um `Observer`, é possível implementar apenas o caso de sucesso com o método `accept()`. No entanto, caso o `Observable` traga algum erro, essa `exception` não será tratada dentro desse `Consumer` e deverá ser tratada em algum outro ponto do fluxo.

Disposables

Através da utilização de um `Observer`, conseguimos nos conectar aos `Observables`, gerando algo que é chamado de `Subscription` ou `Disposable`. Como estamos trabalhando com programação assíncrona, é muito comum se conectar a algo que pode enviar novas informações a qualquer momento. No entanto, em certas situações, utilizamos o comando `dispose()` para remover o `Observer`, como nos trechos a seguir.

```
// É muito comum utilizar Observables dentro dos ViewModels
class ViewModel {
  // Propriedade no escopo da classe para ficar acessível nos métodos
  val disposable: Disposable

  // Em alguma inicialização qualquer
  init {
    val nameChangeObservable = RxTextView.textChanges(etName)

    // Faz a subscription e guardar o Disposable
    disposable = nameChangeObservable.subscribe {
```

```
      // Trabalhar com o evento de mudança do texto do EditText
    }
  }

  // Quando um ViewModel não é mais necessário, o `onCleared()` é
executado:
  override fun onCleared() {
    disposable.dispose() // Limpa todas as Subscriptions
  }
}
```

`CompositeDisposable` : uma forma prática de usar o `Disposable`

Em alguns casos, é possível ter várias `Subscriptions` acontecendo em uma mesma classe. Para facilitar o gerenciamento dos `Diposables` sem precisar de uma propriedade para cada um deles, podemos utilizar o `CompositeDisposable` , que permite colocar todos os `Disposables` em um único lugar e limpá-los todos de uma vez, da seguinte forma:

```
class ViewModel {
  // Criamos o CompositeDisposable
  val composite = CompositeDisposable()

  // Em alguma inicialização qualquer
  init {
    val nameChangeObservable = RxTextView.textChanges(etName)
    val ageChangeObservable = RxTextView.textChanges(etAge)

    // Adicionar as duas Subscriptions na mesma propriedade
    composite += nameChangeObservable.subscribe {
      // ...
    }

    composite += ageChangeObservable.subscribe {
      // ...
    }
  }
```

```
  // Limpa tudo de uma vez
  override fun onCleared() {
    composite.dispose()
  }
}
```

Operators

Como vimos anteriormente, o `Observable` possui uma informação que é observada pelo `Observer`, gerando um `Disposable` que permite remover essa relação. Os `Operators` entram exatamente no meio dessa relação, permitindo que o dado inicial se transforme ao longo do fluxo. Isso facilita trabalhar com fluxos complexos em que muitas etapas são necessárias até chegar à informação final.

Existem diversos operadores disponíveis, sendo que cada um tem funcionalidades muito específicas, como transformar uma informação A em B, filtrar dados indesejados, permitir realizar qualquer ação como exibir os dados recebidos no console, entre outros. Apesar de existirem muitos operadores, conhecer apenas os principais já permite uma utilização satisfatória do RX, uma vez que os mais complexos geralmente simplificam comportamentos que poderiam ser obtidos com operadores mais básicos.

Uma lista completa está disponível na documentação (link a seguir), juntamente a um diagrama que explica cada um. Visualizar os diagramas dos operadores facilita o entendimento, pois alguns sites permitem simular alterações no fluxo em tempo real, demonstrando o que aconteceria na vida real se utilizarmos o operador em questão. Vamos ver alguns exemplos com os operadores mais famosos.

Link para a documentação de operadores do RX:
https://reactivex.io/documentation/operators.html.

Figura 10.3: Documentação de operadores do RX.

Link para testar operadores de forma visual:
https://rxmarbles.com/

Figura 10.4: Teste de operadores de forma visual.

Map

O funcionamento do operador `map` consiste em transformar uma informação recebida em outra qualquer, de acordo com a nossa declaração.

```
Observable.just(1, 2, 3)
     .map { it * 2 } // 2, 4, 6
     .subscribe { println(it) } // 2, 4, 6
```

Nesse exemplo, para cada número recebido, vamos passá-lo pelo `map` e transformá-lo em seu dobro. Como a informação inicial era `1, 2, 3`, o resultado dessa operação será `2, 4, 6`.

Filter

Também podemos adicionar a essa declaração um `filter`, que vai filtrar o resultado e retornar apenas aqueles que desejamos.

```
Observable.just(1, 2, 3)
     .map { it * 2 } // 2, 4, 6
     .filter { it < 6 } // 2, 4
     .subscribe { println(it) } // 2, 4
```

Nesse caso, o filtro vai permitir apenas números passados pelo `map` que são menores do que 6, retornando então `2, 4` como resultado da operação.

FlatMap

Em um primeiro momento, pode parecer que o operador `flatMap` realiza a mesma coisa que o `map`; porém, apesar de serem bem parecidos, a diferença é que o `flatMap` transforma um item recebido em um `Observable`.

Podemos receber um conjunto de objetos de usuários observáveis, como `Observable<User>`, e transformar em uma lista de posts, como `Observable<Post>`, que são obtidos através de uma chamada na API, que retorna um `Observable` e processa os posts logo em sequência.

```
val usersObservable: Observable<User> = getUsersFromApi();
```

```
val posts: Observable<Post> = usersObservable.flatMap {
  it.getUserPostsFromApi() // getUserPosts faz uma chamada na API
}

// Processamos os posts logo em sequência
```

Por exemplo, o `flatMap` recebe 10 usuários de uma única vez e, para cada um deles, acessa a lista de posts, fazendo uma chamada de API para cada um. Os resultados que forem recebidos são processados logo em sequência. Portanto, se chamarmos 10 requisições em ordem, as mais rápidas serão processadas, não importa a ordem em que foram chamadas.

10.4 Implementando o RX no projeto

Agora que conhecemos um pouco mais do RX, chegou a hora de implementá-lo no projeto. O objetivo dessa implementação é facilitar a manipulação das requisições assíncronas da aplicação, sejam requisições em APIs ou em banco de dados local. Para fazer isso, vamos alterar as principais estruturas de dados da aplicação, trabalhando principalmente nas camadas *View Model* e *Repository*.

Primeiro, para adicionar o RX ao projeto, devemos referenciá-lo no `build.gradle` do `app`, com as seguintes linhas:

```
//...
dependencies {
  //...

  // RX
  implementation("io.reactivex.rxjava3:rxjava:$rxVersion")
}
```

Utilizamos a biblioteca `rxjava` apenas neste capítulo, visto que é possível acessar as declarações do RX através de outras bibliotecas que veremos no próximo capítulo, como em conjunto com o Android Room (`room-rxjava3`) e com o Retrofit (`adapter-rxjava3`).

Após realizar a sincronização no Gradle clicando no botão `Sync now` , vamos preparar o `Observable` que emitirá um evento contendo a lista de criaturas disponíveis na aplicação. Para isso, abra o arquivo `CreaturesRepository` e procure pela propriedade `creatures` , do tipo `List<Creature>` . Para transformá-la em um `Observable` , basta envolver o tipo real da informação (`List<Creature>`) no tipo do `Observable` , ficando `Observable<List<Creature>>` , como a seguir.

```
// ...
import io.reactivex.rxjava3.core.Observable
// ...
class CreaturesRepository @Inject constructor() {
  val creatures: Observable<List<Creature>>
  // ...
}
```

Agora que a propriedade `creatures` é do tipo `Observable` , precisamos atualizar sua inicialização. Dentro do bloco `init { }` , vamos criar o `Observable` utilizando a declaração `Observable.create()` , como mostrado anteriormente neste capítulo.

```
// ...
class CreaturesRepository @Inject constructor() {
  val creatures: Observable<List<Creature>>

  init {
    // ...
    creatures = Observable.just(
      listOf( /* ... */ )
    )
    //...
```

Modificar a camada *Model* sempre tem um alto impacto, pois estamos alterando a principal estrutura da aplicação. Como a propriedade `creatures` é utilizada no método `CreaturesRepository.findCreature()`, no `UserRepository.allCreatures` e no `UserRepository.chooseCreature()`, precisamos realizar os ajustes em cada um deles.

Primeiro, vamos adaptar o `CreaturesRepository.findCreature()`. Atualmente, o método não possui um tipo de retorno declarado, mas está retornando um objeto do tipo `Creature`. No entanto, como estamos trabalhando com `Observable`, vamos retornar um `Observable<Creature>`, obtido a partir do `Observable<List<Creature>>`, utilizando o número da criatura recebido via parâmetro.

Um dos grandes potenciais do RX é a possibilidade de transformar informações sem se preocupar se elas são síncronas ou assíncronas, tratando todas como um único fluxo de dados. No próximo capítulo, vamos precisar de execuções que buscam dados na internet, por exemplo. Para esses casos, o RX permite uma fácil inserção de uma chamada à API no meio de um fluxo de carregamento de dados, sem precisar lidar com toda a dor de cabeça que é trabalhar com chamadas assíncronas.

Como dito anteriormente, para adaptar o `findCreature`, vamos declarar um fluxo de dados que transforma a propriedade `creatures` do tipo `Observable<List<Creature>>` em um `Observable<Creature>`. Para isso, vamos utilizar o operador `map`, que é destinado a transformar uma informação ao longo do fluxo de dados.

```
// ...
class CreaturesRepository @Inject constructor() {
  // ...
  fun findCreature(number: Int) =
      creatures.map { list ->
        list.find { it.number == number }
      }
}
```

Como o RX não consegue determinar se a operação executada dentro do `map` funcionará ou não, o retorno é `nullable` por padrão, como mostra a imagem a seguir.

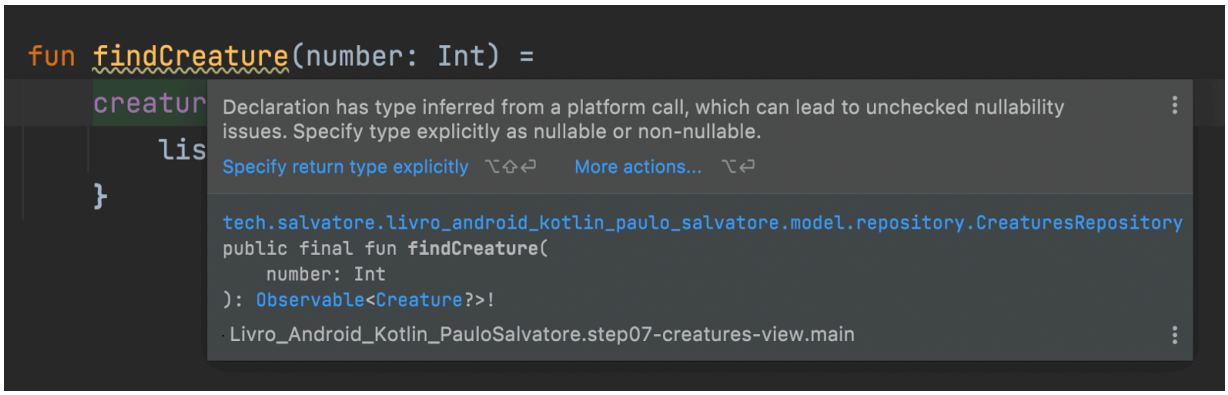


Figura 10.5: Associação automática de tipo define como nullable.

Para garantir que não vamos precisar tratar valores nulos durante a utilização do `Observable`, é interessante colocar o tipo de retorno do método de forma explícita como `Observable<Creature>`:

```
// ...
class CreaturesRepository @Inject constructor() {
  // ...
  fun findCreature(number: Int): Observable<Creature> =
      // ...
}
```

Com o `CreaturesRepository` adaptado, vamos realizar os ajustes no `UserRepository`, começando pela propriedade `allCreatures`. A forma mais fácil de adaptar ou criar uma propriedade `Observable` é adicionando o tipo, para que o Android Studio auxilie nesse processo. Nesse caso, a propriedade era do tipo `List<Creature>` e será transformada em `Observable<List<Creature>>`:

```
// ...
import io.reactivex.rxjava3.core.Observable

class UserRepository @Inject constructor(
  // ...
```

```
val allCreatures: Observable<List<Creature>>
  get() = creaturesRepository.creatures.map {
    val isKnown = user.creatures.any { creatureOwnByUser ->
      creatureOwnByUser.number == it.number
    }

    it.copy(
      name = if (isKnown) it.name else "?????",
      isKnown = isKnown
    )
  }
      // ...
```

Agora que adicionamos o tipo, o Android Studio sinaliza que o retorno do `map` não atende ao tipo `List<Creature>`, pois está retornando uma informação do tipo `Unit`, ou seja, não está retornando nada.

> `Unit` é o equivalente ao `void` do Java. Quando um método no Android não possui retorno, a informação retornada é sempre do tipo `Unit`.

Esse erro ocorre pois a declaração `it.copy` está quebrando, uma vez que o auxiliar `it` possui a lista de criaturas, mas antes possuía cada criatura em si. Algo muito fácil de confundir é a utilização do *operator* `map` de listas do Kotlin/Java e o operador `map` do RX. Apesar de ambos os operadores transformarem uma informação em outra, eles possuem implementações bem diferentes.

Antes de resolver o problema, vamos relembrar que o objetivo da propriedade `allCreatures` dentro do `UserRepository` é trazer a lista de criaturas a partir da perspectiva do usuário, informando quais criaturas ele conhece ou não, para que a UI saiba quando exibir a silhueta ou a imagem oficial da criatura.

Agora que introduzimos o `Observable` para a lista de criaturas do `CreaturesRepository`, a propriedade `allCreatures` deve transformar

uma `List<Creature>` em outra `List<Creature>`, mas a partir da perspectiva do usuário. Para isso, vamos usar o operador `map` do `Observable` e, dentro do fluxo de dados, manter a mesma utilização do operador `map` da lista, como estávamos fazendo anteriormente.

```
// ...
class UserRepository @Inject constructor(
  // ...
  val allCreatures: Observable<List<Creature>>
    get() = creaturesRepository.creatures.map { list ->
      list.map {
        // ...
      }
    }
    // ...
```

Com a propriedade `allCreatures` corrigida, vamos adaptar o método `chooseCreature()`, que realiza 5 ações:

1. Checa se o usuário pode receber novas criaturas, através da propriedade `hasCreaturesAvailable`;
2. Indica que o usuário não pode mais receber novas criaturas, atualizando o valor da propriedade `hasCreaturesAvailable`;
3. Seleciona uma criatura aleatória do `CreaturesRepository.creatures`;
4. Adiciona a criatura à lista de criaturas do usuário;
5. Atualiza o `LiveData onChooseCreature` com a criatura que acabou de ser adicionada ao usuário.

Até então, como a aplicação não realizava nenhuma chamada assíncrona, como enviar esse dado para a API ou para um banco de dados local, trabalhar com listas e objetos é algo bem tranquilo de se fazer. No entanto, visto que estamos preparando a aplicação para trabalhar com fluxos de dados utilizando o RX, que facilita a integração entre fluxos síncronos e assíncronos, vamos transformar toda a implementação desse método para utilizar os princípios do RX da melhor forma.

Como o RX sugere a utilização de uma programação funcional reativa, vamos trabalhar com pequenas partes do fluxo, de modo que fique claro o que estamos fazendo em cada uma das etapas e também facilite a manutenção desse método no futuro. Para isso, iniciamos o fluxo a partir de um `Observable`, combinamos `Operators` que moldam o fluxo ao longo do processo e retornamos um `Observable` ao final, para que ele seja observado por quem tiver interesse.

Primeiro, vamos atualizar o tipo de retorno do método:

```
fun chooseCreature(): Observable<Creature> =
```

Depois, iniciamos o fluxo de dados com o `Observable`. Como o `user` continua sendo apenas um objeto (podemos adaptá-lo no futuro, caso formos salvar o `user` em um banco de dados, por exemplo), criamos um `Observable<User>` através da declaração `Observable.just()`.

```
fun chooseCreature(): Observable<Creature> =
  Observable.just(user)
```

O fluxo de dados deve avançar apenas caso a propriedade `hasCreatureAvailable` do `user` tenha `true` como valor. Para isso, podemos utilizar o operador `filter`, que filtra os eventos e processa apenas os que retornarem `true`.

```
fun chooseCreature(): Observable<Creature> =
  Observable.just(user)
    .filter {
      it.hasCreatureAvailable
    }
```

Atualizamos o `user`, informando que não há mais criaturas disponíveis. Dentro de um `Observable`, sempre que quisermos apenas executar uma ação, podemos utilizar o operador `doOnNext { }`.

Operadores que apenas executam algo, sem se importar com o retorno, costumam começar com a palavra `do`.

```
fun chooseCreature(): Observable<Creature> =
  Observable.just(user)
    // ...
    .doOnNext {
      it.hasCreatureAvailable = false
    }
```

Agora que o usuário foi devidamente checado e atualizado, vamos transformar o fluxo atual em um fluxo com a lista de criaturas disponível em `CreaturesRepository.creatures`.

```
fun chooseCreature(): Observable<Creature> =
  Observable.just(user)
    // ...
    .doOnNext {
      it.hasCreatureAvailable = false
    }
    .map {
      creaturesRepository.creatures
    }
```

Ao transformar um `Observable` em outro, o RX não entende de forma automática que queremos trabalhar com os dados que estão dentro do outro fluxo. Na imagem a seguir, é possível notar através das dicas do Android Studio que o tipo de `it` foi transformado de `User!` para `Observable<List<Creature>>!`.

Figura 10.6: Transformando um Observable em outro.

Para sinalizar que o fluxo deve avançar com a lista de criaturas em si, devemos utilizar o operador `flatMap` que, além de transformar o fluxo, garante que o `Observable` será resolvido antes que o fluxo prossiga.

```
fun chooseCreature(): Observable<Creature> =
  Observable.just(user)
    // ...
    .doOnNext {
      it.hasCreatureAvailable = false
    }
    .flatMap {
```

```
      creaturesRepository.creatures
    }
```

Agora que temos a lista de criaturas, transformamos o fluxo em apenas uma única criatura, que será obtida através da declaração `random()`, como foi feito anteriormente.

```
fun chooseCreature(): Observable<Creature> =
  Observable.just(user)
    // ...
    .map {
      val randomCreature = it.random()
      randomCreature
    }
```

Com a criatura aleatória disponível no fluxo, criamos um novo bloco de `doOnNext` para adicioná-la à lista de criaturas do usuário.

```
fun chooseCreature(): Observable<Creature> =
  Observable.just(user)
    // ...
    .doOnNext {
      user.creatures.add(it)
    }
```

Por fim, atualizamos o `LiveData` `onChooseCreature` com a criatura recém-adicionada.

```
fun chooseCreature(): Observable<Creature> =
  Observable.just(user)
    // ...
    .doOnNext {
      _onChooseCreature.value = it
    }
```

Observando os dados na camada *View Model*

Lembre-se de que todo `Observable` precisa de um `Observer` para iniciar o processamento do fluxo de dados. Portanto, precisamos

atualizar as implementações na camada *View Model* para criar os `Observers` e colocar os dados na propriedade `LiveData`.

Vamos começar pelo `CreaturesViewModel`. Atualmente, a propriedade `creatures` é inicializada com o uso do valor de `UserRepository.allCreatures` na íntegra. No entanto, como agora temos um `Observable` nessa propriedade, devemos criar o `Observer` e usar o valor recebido para atualizar o `LiveData`. A forma mais fácil de fazer isso dentro de um *View Model* é criar o `subscribe` dentro do bloco `init { }`, da seguinte forma.

```
// ...
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.domain.Creature

class CreaturesViewModel @Inject constructor(
    // ...
  val creatures = MutableLiveData<List<Creature>>()

  init {
    userRepository.allCreatures.subscribe {
      creatures.value = it
    }
  }
  // ...
}
```

Note que o Android Studio exibe um aviso, informando que o retorno do `subscribe` não está sendo utilizado, como é possível ver na imagem a seguir.

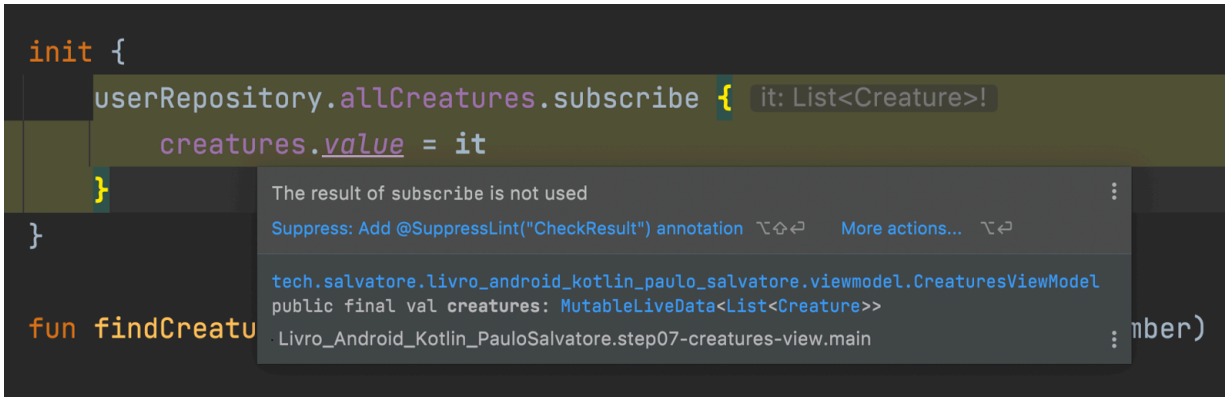


Figura 10.7: Retorno do subscribe não usado.

Essa dica é fundamental, pois um dos erros mais comuns com a utilização do RX é esquecer-se de finalizar uma *subscription* em um `Observable`. Como mencionado anteriormente, existem duas formas de realizar isso, uma com o `.dispose()` e outra com o `CompositeDisposable()`.

Para esconder o aviso e finalizar o ajuste da propriedade `creatures` do `CreaturesViewModel`, vamos implementar o `CompositeDisposable()`. Primeiro, criamos a propriedade. Depois, garantimos que, durante o `onCleared()` do `ViewModel`, o `dispose()` seja executado corretamente.

```
import io.reactivex.java3.disposables.CompositeDisposable

class CreaturesViewModel @Inject constructor(
  // ...
  private val composite = CompositeDisposable()

  init {
    userRepository.allCreatures.subscribe { /* ... */ }
    // ...
  }

  override fun onCleared() {
    composite.dispose()
  }

  // ...
```

Com o `CompositeDisposable` preparado, vamos adicionar o resultado do `subscribe` ao `composite`, através da declaração `composite.add()`:

```
// ...
class CreaturesViewModel @Inject constructor(
  // ...
  init {
    composite.add(userRepository.allCreatures.subscribe {
      creatures.value = it
    })
    // ...
```

Uma forma de simplificar essa declaração é com o método `plusAssign` do Kotlin, que permite a utilização do operador `+=` para qualquer objeto, executando um trecho de código completamente customizável. Para integrar essa implementação com o `CompositeDisposable`, vamos criar um arquivo de extensão.

Primeiro, dentro do pacote principal da aplicação, crie um pacote chamado `extensions.rx`. Dentro dele, crie um arquivo `Object` chamado `CompositeDisposableExtensions.kt`, adicionando o seguinte conteúdo:

```
package tech.salvatore.livro_android_kotlin_paulo_salvatore.extensions.rx

import io.reactivex.rxjava3.disposables.CompositeDisposable
import io.reactivex.rxjava3.disposables.Disposable

object CompositeDisposableExtensions {
    operator fun CompositeDisposable.plusAssign(disposable: Disposable) {
        this.add(disposable)
    }
}
```

Com isso, podemos simplificar a declaração do `CreaturesViewModel` para o trecho a seguir:

```
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.extensions.rx.Composit
eDisposableExtensions.plusAssign
```

```
class CreaturesViewModel @Inject constructor(
  // ...
  init {
    composite += userRepository.allCreatures.subscribe { /* ... */ }
    // ...
```

Agora que o `CreaturesViewModel` está preparado, vamos preparar também o `CreatureViewModel`, ajustando o método `loadCreature()`. Nada de diferente nessa implementação, é exatamente o mesmo processo feito anteriormente. O código atualizado ficará como o seguinte:

```
// ...
import io.reactivex.rxjava3.disposables.CompositeDisposable
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.extensions.rx.Composit
eDisposableExtensions.plusAssign

@HiltViewModel
class CreatureViewModel @Inject constructor(
  // ...

  private val composite = CompositeDisposable()

  fun loadCreature(number: Int) {
    composite += creaturesRepository.findCreature(number).subscribe {
      _creature.value = it
    }
  }

  override fun onCleared() {
    composite.dispose()
  }
}
```

Para finalizar a camada *View Model*, vamos realizar o mesmo procedimento no arquivo `UserViewModel`.

```
// ...
import android.util.Log
import io.reactivex.rxjava3.disposables.CompositeDisposable
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.extensions.rx.Composit
eDisposableExtensions.plusAssign

class UserViewModel @Inject constructor(
  // ...
  private val composite = CompositeDisposable()

  fun chooseCreature() {
    composite += userRepository.chooseCreature().subscribe {
      Log.d("CHOOSE_CREATURE", it.toString())
    }
  }

  override fun onCleared() {
    composite.dispose()
  }
}
```

O último local que precisamos arrumar é o arquivo `CreatureAddedFragment`, que se conecta diretamente ao método `findCreature` do `CreatureViewModel`, utilizando o argumento `creatureId` recebido via *navigation*. Apesar de ser possível executar métodos que retornam `Observables` a partir de classes que possuem ciclo de vida (como `Activities` e `Fragments`), esta não é uma boa prática, pois o gerenciamento da *subscription* e do *disposable* dão um pouco de trabalho em classes muito vulneráveis à atividade do usuário. Além disso, todo esse gerenciamento já é fornecido pelo `LiveData` que recebe o `LifecycleOwner` e faz tudo de forma automática.

Portanto, vamos encurtar o caminho entre `CreatureAddedFragment` e o `ViewModel`. Atualmente, a classe `CreatureAddedFragment` está se conectando ao `CreaturesViewModel` para utilizar o método `findCreature()`. No entanto, temos um método no `CreatureViewModel` que realiza exatamente o carregamento da criatura a partir de um

número, armazenando a criatura encontrada em uma propriedade do tipo `LiveData`.

A implementação do `CreatureAddedFragment` seguirá estes passos:

- Atualizar a propriedade `viewModel` para `CreatureViewModel`;
- Adicionar a propriedade `viewModel` direto no arquivo XML, para inseri-la no mecanismo de *Data Binding*;
- Remover a propriedade `creature` do arquivo XML, pois passaremos a utilizar diretamente o `viewModel`;
- Inserir o `ViewModel` como valor da propriedade `viewModel` do `binding`;
- Executar o método `loadCreature()`, informando o `creatureId` recebido via argumento de *navigation*.

Primeiro, substituímos a propriedade `creature` pelo `viewModel`, direto no arquivo `creature_added_fragment.xml`, colocando o tipo como `CreatureViewModel`.

```
<?xml version="1.0" encoding="utf-8"?>
<layout ...>
  <data>
    <variable
      name="viewModel"

type="tech.salvatore.livro_android_kotlin_paulo_salvatore.viewmodel.Creatu
reViewModel" />
  </data>

  <androidx.constraintlayout.widget.ConstraintLayout ...>
  <!-- ... -->
```

Ao remover a propriedade `creature`, alguns elementos do layout passam a acusar alguns erros, justamente por não encontrar a propriedade removida. Basta fazê-los acessar a propriedade `viewModel` antes de acessar a propriedade `creature`.

```
<?xml version="1.0" encoding="utf-8"?>
<layout ...>
```

```
  <data>
    <!-- ... -->
  </data>

  <androidx.constraintlayout.widget.ConstraintLayout ...>
    <!-- ... -->
    <androidx.constraintlayout.widget.ConstraintLayout ...>
      <TextView
                android:id="@+id/tvTitle"
                ...

android:text="@{@string/creature_hatched(viewModel.creature.name)}"
                ... />
    </androidx.constraintlayout.widget.ConstraintLayout>

    <ImageView
               android:id="@+id/ivCreature"
               ...
               app:imageUrl="@{viewModel.creature.imageUrl}"
               ... />

  </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Com o XML devidamente atualizado, vamos para a classe `CreatureAddedFragment` para mudar o tipo da propriedade `viewModel` para `CreatureViewModel`.

```
// ...
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.viewmodel.CreatureView
Model

class CreatureAddedFragment : Fragment() {
  // ...
  private val viewModel: CreatureViewModel by viewModels()
  // ...
```

Agora que a propriedade está atualizada, vamos inserir o valor obtido no *Data Binding* dentro do método `onViewCreated()`. Além

disso, executamos o método `viewModel.loadCreature()` para carregar a propriedade `LiveData`, que é utilizada pelo layout XML. Não se esqueça de remover a declaração antiga do `viewModel.findCreature()` e `binding,creature`. O conteúdo do método `onViewCreated()` ficará assim:

```
// ...
class CreatureAddedFragment : Fragment() {
  // ...
  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    // Set ViewModel
    binding.viewModel = viewModel

    // Load creature by number
    val safeArgs: CreatureAddedFragmentArgs by navArgs()
    val creatureId = safeArgs.creatureNumber
    viewModel.loadCreature(creatureId)

    // See creature in list
    val btAddCreature = view.findViewById<Button>(R.id.btAddCreature)
    btAddCreature.setOnClickListener {
      findNavController().popBackStack()
    }
  }
}
```

Pronto! Rode a aplicação e teste o fluxo de adição da criatura, a visualização na lista e a visualização individual da criatura.

Apesar de a implementação deste capítulo não adicionar novos comportamentos à aplicação em si, acabamos de adicionar algo muito poderoso, que é tornar chamadas assíncronas para API ou banco de dados de uma forma muito mais fácil e sem precisar mexer em nenhum código fora da camada *Repository*, que já está toda conectada.

Conclusão

Neste capítulo, entendemos a diferença entre programação síncrona e assíncrona, mostrando as bases do RX para facilitar o trabalho com esse tipo de implementação. Além disso, vimos como implementar o RX no projeto, deixando a aplicação preparada para a integração entre o `Repository` e uma fonte de dados como API ou banco de dados.

No próximo capítulo, vamos implementar a chamada à API, também conhecida como back-end, para carregar a lista de criaturas que estarão disponíveis na aplicação. Isso permitirá controlar quais criaturas iniciais o aplicativo terá, sem a necessidade de ter que atualizar a versão do aplicativo para realizar o lançamento de novas criaturas.

Por último, mas não menos importante, chegou a hora de revelar que o nome do DevMon #10 é o **NestJS**, inspirado no framework de back-end para Node.js que vem crescendo em popularidade pelo fato de ter uma arquitetura muito bem pensada e possuir uma integração excelente com as principais bibliotecas JavaScript do cenário. Se estiver pensando em construir uma aplicação mais robusta de back-end utilizando TypeScript ou JavaScript, o NestJS pode ajudar nesse processo.

URL da imagem: https://i.imgur.com/1IOVD1l.png

Figura 10.8: O DevMon #10 se chama NestJS!

Capítulo 11
Integrando com back-end

> Caso queira visualizar as implementações completas deste capítulo, acesse o módulo `cap11-api-backend`.

Até este capítulo, desenvolvemos a maior parte da aplicação para funcionar de forma offline, utilizando a internet apenas para carregar as imagens dos DevMon. No entanto, quando o assunto é o conteúdo da aplicação, é interessante ter algo que seja possível controlar de maneira remota, sem a necessidade de recompilar o aplicativo e enviar uma nova versão para a loja toda vez que o conteúdo mudar.

A lista de criaturas, por exemplo, é algo que deve ser controlado de maneira remota. Para isso, precisamos de um **back-end**, também conhecido como **API**, que esteja preparado para receber solicitações através da internet e retornar os resultados desejados. Para este capítulo, foi desenvolvido um back-end utilizando o framework NestJS, que utiliza TypeScript como linguagem de programação.

Saber criar um back-end não é um requisito para quem desenvolve aplicações no Android, basta conhecer como funcionam as rotas, também conhecidas como *endpoints*, e saber analisar as estruturas de dados. No entanto, caso você queira entender como o back-end da nossa aplicação funciona, o código está disponível no GitHub, através dos links a seguir.

Repositório: https://github.com/FabricaDeSinapse/devmon-api

Figura 11.1: Repositório da API Back-end no GitHub.

Back-end: https://devmon-api.onrender.com

Figura 11.2: Deploy da API Back-end.

Antes da implementação, como de costume, vamos falar sobre as criaturas que serão reveladas neste capítulo, o DevMon #11 e sua evolução, o DevMon #12.

O nome do DevMon #11 é em homenagem a uma tecnologia de banco de dados relacionais extremamente difundida, e é um requisito para qualquer pessoa que desenvolve aplicações e precisa armazenar dados de alguma forma. Vários bancos de dados utilizam essa tecnologia de uma forma bem parecida, apesar de possuir dialetos diferentes e característicos dependendo do que estiver sendo utilizado.

Já o DevMon #12 é nomeado em homenagem a uma outra tecnologia de banco de dados, que serve como uma ótima alternativa aos bancos relacionais, principalmente por apresentar formas diferentes de estruturação dos dados, trazendo soluções muito mais flexíveis em relação ao *schema*, ou muito mais leves e mais rápidas. No entanto, essa tecnologia acaba servindo mais como um termo para designar modelos de bancos de dados que não se encaixam completamente no modelo relacional.

E aí, já sabe quais são?

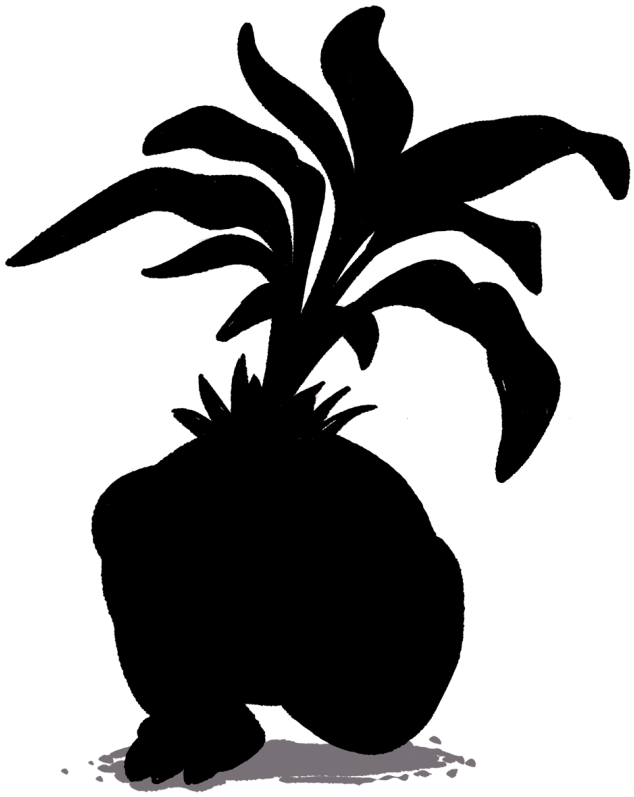


Figura 11.3: Qual é o DevMon #11?

Figura 11.4: Qual é o DevMon #12?

11.1 Networking

Networking no Android é a capacidade de se comunicar com algum serviço disponível na web, recebendo dados estruturados em formatos como JSON, XML, entre outros. Na prática, o dado vindo do back-end é uma `string`, pois representa uma sequência de caracteres. Para indicar qual é o formato do dado enviado, o back-end envia um cabeçalho na resposta indicando a estrutura — por exemplo, `application/json` para dados JSON ou `text/html` para um código HTML.

Apesar de muito comum, carregar informações da internet não é uma tarefa tão simples, afinal, muitos problemas podem ocorrer,

como perda de conexão, lentidão no processo, queda de sinal, entre diversos outros motivos.

Neste capítulo, entenderemos a principal forma de comunicação pela web, utilizando o protocolo HTTP e recebendo dados estruturados em JSON (*Java Script Object Notation*, que entenderemos mais à frente). Veremos também os principais meios de transformar textos JSON em objetos que podem ser utilizados na programação, além de conhecer bibliotecas que facilitam essa implementação no Android. Vale ressaltar que toda requisição na web deve ser feita em uma `thread` separada da *UI Thread*; portanto, vamos utilizar o RX para facilitar o trabalho com execuções assíncronas.

11.2 Retrofit

A biblioteca Retrofit foi desenvolvida e é mantida pela Square, além de ser uma das mais populares no ecossistema dev Android, estando presente na maioria dos projetos que precisam fazer requisições na web. Ela possui um uso extensivo de `annotations`, que são capazes de simplificar a implementação dos serviços que realizam operações no back-end.

Para implementar o Retrofit, precisamos realizar os seguintes passos:

1. Importar a biblioteca no `build.gradle` da nossa aplicação;
2. Criar pelo menos uma interface que tenha o papel de `Service`. Essa interface, que será implementada pelo Retrofit através de geração automática de código, contém as rotas do back-end, incluindo:
    - Verbos do protocolo HTTP: `GET`, `POST`, `PUT`, `DELETE` etc. Obtemos essa informação através da documentação do back-end.

    - *Endpoint*. No caso da URL `https://devmon-api.onrender.com/creature`, por exemplo, o *endpoint* é `creature`.
3. Instância do Retrofit. Criada através da declaração `Retrofit.Builder().build()`.
4. Uma classe que gerencie a instância do Retrofit e disponibilize o acesso aos `Services` declarados. Apesar de ser possível criar a instância do Retrofit em qualquer código, ter um arquivo único para realizar esse gerenciamento facilita a manutenção dos códigos de acesso ao back-end.

Trabalhar com requisições na web é algo relativamente complexo de ser realizado no Android. No entanto, o Retrofit consegue fornecer um kit de ferramentas poderoso para interagir com o protocolo HTTP, disponibilizando todos os recursos necessários para criar conexões, receber os dados e processá-los, incluindo mecanismos para tratamento de erros.

Como dito anteriormente, os dados recebidos do back-end são `strings` que seguem uma estrutura específica, como o JSON. Na prática, o back-end transformou um objeto da programação em texto, em um processo conhecido como *serialização*. A partir disso, a `string` foi enviada para o Android através do protocolo HTTP. Para conseguir trabalhar com os dados que foram serializados, precisamos transformá-los novamente em objetos da programação, em um processo chamado *desserialização*. Apesar de o Retrofit não se propor a resolver esse problema, é disponibilizada uma integração fácil com bibliotecas que resolvem isso, como o GSON, da Google, ou o Moshi, da Square.

11.3 JSON

As duas formas mais comuns de estruturação de dados são JSON e XML. O JSON (*Java Script Object Notation*) é uma forma de

estruturação de dados muito comum para trocar informações entre diversas aplicações, independente da linguagem de programação em que ela é feita. Inspirado na declaração de objetos da linguagem JavaScript, o JSON é considerado uma das formas mais leves de estruturar uma informação, com grande vantagem em relação ao XML. Por exemplo, para representar um objeto de dados de uma pessoa e seus interesses, temos as seguintes versões em XML e JSON.

- **XML: 211 caracteres**

```xml
<pessoa>
  <nome>Paulo Salvatore</nome>
  <interesses>
      <interesse>Kotlin</interesse>
      <interesse>TypeScript</interesse>
      <interesse>Unity</interesse>
      <interesse>C#</interesse>
  </interesses>
</pessoa>
```

- **JSON: 88 caracteres**

```json
{
  "nome": "Paulo Salvatore",
  "interesses": ["Kotlin", "TypeScript", "Unity", "C#"]
}
```

No XML, cada tag que foi aberta deve ser fechada, o que a faz repetir o nome usado na abertura e gastar muito mais *bytes* apenas para abrir e fechar conteúdos. Isso não acontece no JSON, já que a estruturação é organizada utilizando símbolos como aspas, chaves e colchetes. Além disso, o formato JSON permite armazenar de forma mais fácil alguns tipos primitivos de dados, diferenciando `string` de informações como `number` e `boolean`, enquanto o XML armazena tudo como `string` por padrão.

JSON Viewer

Quando abrimos URLs de servidores back-end que exibem dados em JSON, é muito difícil interpretar a informação, pois nem todo navegador estrutura a informação de forma automática. Para resolver esse problema, é possível instalar extensões como o JSON Viewer, que transforma a visualização para facilitar tanto o entendimento quanto a manipulação da informação.

No Google Chrome, por exemplo, a extensão pode ser baixada diretamente da loja, através do link a seguir: https://chrome.google.com/webstore/detail/json-viewer/gbmdgpbipfallnflgajpaliibnhdgobh

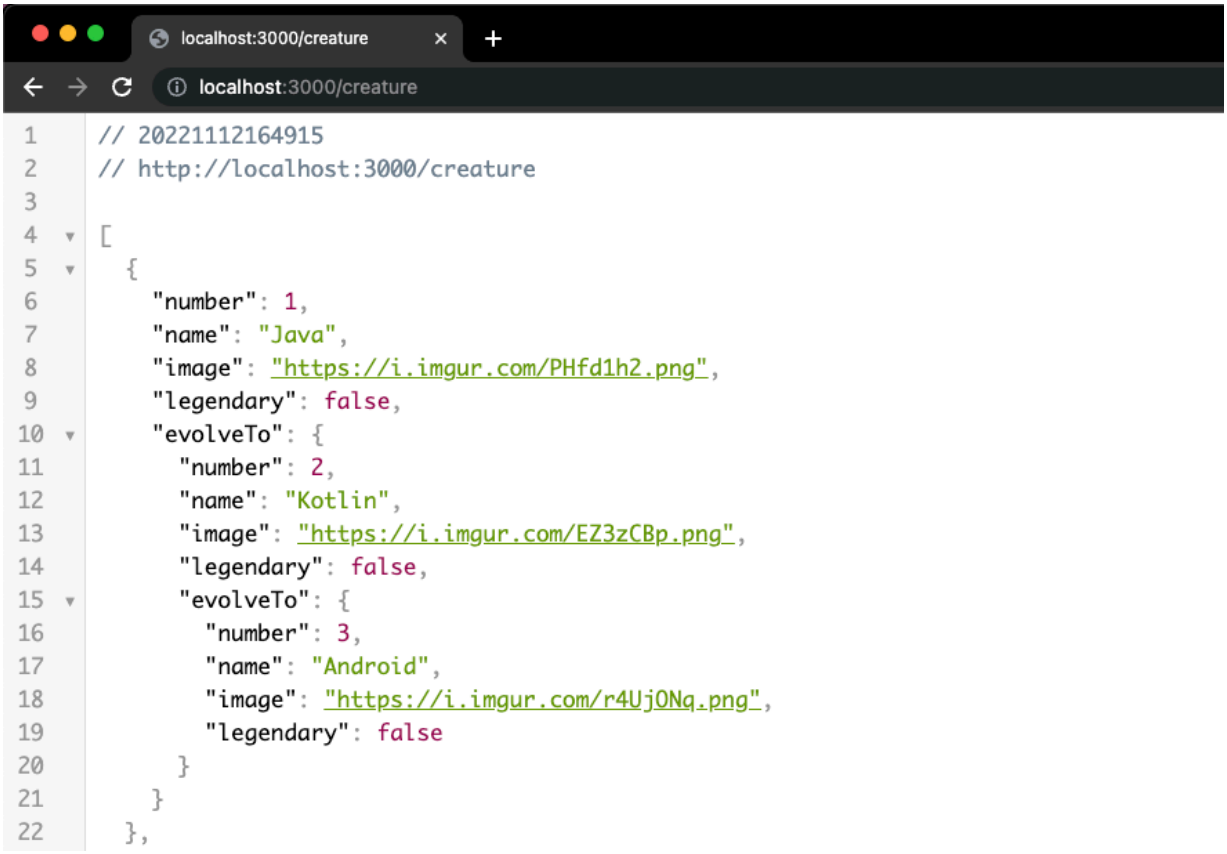


Figura 11.5: Exibição de JSON formatada pelo JSON Viewer.

Serialização

Agora que sabemos como funciona a estrutura de um JSON e temos ferramentas para interpretá-la de forma mais clara, precisamos saber como fazer a programação entendê-la, para que seja possível trabalhar com os dados recebidos nesse formato. Para isso, precisamos conhecer o processo de serialização. Enquanto serializar um dado significa transformar a informação que está na memória RAM do computador em um texto estruturado, desserializá-lo significa pegar esse texto e inseri-lo novamente na memória RAM.

Dependendo da linguagem de programação, esse processo pode ser mais simples ou mais complicado. Em linguagens fortemente tipadas, como o Java, temos algumas características mais robustas de tipo de variáveis e estruturação de conteúdo. Por conta disso, o processo de desserialização deve conhecer o tipo de toda a informação presente, antes de poder inseri-la na memória. Utilizando o `JSONParser`, podemos realizar esse processo, como mostra o exemplo a seguir.

```
val jsonData = "{\"colors\":[{\"color\":\"red\",\"hex\":\"#FF0\"},
{\"color\":\"white\",\"hex\":\"#FFF\"}]}"
val reader = JSONObject(jsonData)
val colors = reader.getJSONArray("colors")
val red = colors.getJSONObject(0)
val redHex = red.getString("hex")
// redHex = "#FF0"
```

No entanto, realizar esse processo utilizando as soluções mais simples como `JSONParser` acaba sendo muito complexo e trabalhoso. Para facilitar a manipulação de JSONs, existem algumas bibliotecas, como o GSON (Google), que possuem integração direta com o Retrofit.

11.4 Implementando no projeto

Chegou a tão esperada hora de colocarmos a mão na massa e integrar o back-end ao nosso projeto, vamos lá? Para começar, precisamos garantir que tanto o Retrofit quanto o GSON estão devidamente importados no arquivo `build.gradle` do nosso projeto. Para isso, adicionamos as seguintes linhas:

```
// Retrofit
implementation("com.squareup.retrofit2:converter-gson:$retrofitVersion")
implementation("com.squareup.retrofit2:retrofit:$retrofitVersion")
```

Para aplicações que queiram utilizar o Retrofit em conjunto com o RX, também é necessário importar um pacote extra do Retrofit, o `adapter-rxjava3`, da seguinte forma:

```
// Retrofit
// ...
implementation("com.squareup.retrofit2:adapter-rxjava3:$retrofitVersion")
```

> Agora que temos o RXJava3 vindo do Retrofit, não precisamos mais declarar o RX separadamente, como estávamos fazendo até então. Portanto, remova a declaração `implementation("io.reactivex.rxjava2:rxjava:$rxVersion")` do seu arquivo `build.gradle`.

Conhecendo a API

Para implementar a chamada de API, primeiro precisamos conhecer como interagir com o *endpoint*. Para isso, vamos abrir a URL `https://devmon-api.onrender.com/creature` e analisar a estrutura de dados:


```
[
  {
    "number": 1,
    "name": "Java",
    "image": "https://i.imgur.com/PHfd1h2.png",
    "legendary": false,
```

```
    "evolveTo": {
      "number": 2,
      "name": "Kotlin",
      "image": "https://i.imgur.com/EZ3zCBp.png",
      "legendary": false,
      "evolveTo": {
        "number": 3,
        "name": "Android",
        "image": "https://i.imgur.com/r4UjONq.png",
        "legendary": false
      }
    }
  },
  {
    "number": 2,{
    "number": 2,
    "name": "Kotlin",
    "image": "https://i.imgur.com/EZ3zCBp.png",
    "legendary": false,
    "evolveTo": {
      "number": 3,
      "name": "Android",
      "image": "https://i.imgur.com/r4UjONq.png",
      "legendary": false,
      "evolveTo": null
    }
  },
  // ...
]
```

Como o primeiro nível do JSON é composto por `[` (colchete), temos uma lista de elementos. Para os elementos da lista, temos uma `{` (chave), que é o símbolo para objetos. Para analisar a estrutura de objetos, precisamos sempre esperar por uma propriedade que é composta por `key` (chave) e `value` (valor).

- A `key` é uma `string`, portanto deve ficar dentro de aspas duplas `""`;
- O `value` deve ser um dos tipos permitidos:

    - `boolean`: `true` ou `false`;
    - `string`: `"Texto dentro de aspas duplas"`;
    - `number`: `5` ou `5.5`;
    - `array`: `[ 1, 2, 3 ]`, sendo que os valores podem assumir qualquer tipo;
    - `object`: `{ "key": "value", "key2": "value2" }`;
    - `null`: quando o valor é indefinido.

Após analisar as estruturas de dados das entidades, vamos convertê-las para as estruturas equivalentes do Java/Kotlin:

- `boolean` deve virar `Boolean`;
- `string` deve virar `String`;
- `number` deve virar `Int` ou `Long` (número inteiros), `Float` ou `Double` (números com casas decimais);
- `array` deve virar `Array` ou `List`;
- `object` deve virar uma classe, por se tratar de uma estrutura mais complexa do que os tipos primitivos;
- `null` deve assumir que a propriedade aceita nulos, ou seja, é uma propriedade `Nullable`, representada por `?` no Kotlin.

Aplicando as regras mencionadas, podemos dizer que o *endpoint* `creature` retorna uma informação que é do tipo `List` de uma estrutura customizada de dados. Como precisamos de uma classe para representar essa estrutura, podemos nomeá-la como `CreatureApiResponse`.

Para definir as propriedades da classe `CreatureApiResponse`, precisamos analisar o JSON:

```
{
  "number": 1,
  "name": "Java",
  "image": "https://i.imgur.com/PHfd1h2.png",
  "legendary": false,
  "evolveTo": {
    "number": 2
    // ...
```

```
  }
}
```

Temos as propriedades `number`, `name`, `image`, `legendary` e `evolveTo`. Note que apenas a propriedade `number` representa um número, que aparentemente é um número inteiro e, portanto, vamos utilizar o tipo `Int`. Para as propriedades `name` e `image`, vamos utilizar o tipo `String`. Já a propriedade `legendary` é do tipo `Boolean`.

> Como o tipo `Int` suporta números entre -2.147.483.647 e 2.147.483.647, se assumirmos que o número de criaturas da API será maior do que 2 bilhões de registros, podemos considerar a utilização do tipo `Long`, que aceita uma quantidade muito maior de números.

Por fim, a propriedade `evolveTo` é um `object`. Normalmente, uma nova classe seria necessária, pois objetos são estruturas de JSON customizadas, que só podem ser representadas por classes no Java/Kotlin. No entanto, a estrutura do `evolveTo` aparenta ser exatamente a mesma da `CreatureApiResponse`, com a única diferença de que nem sempre uma criatura possui o campo `evolveTo`, algo que pode ser observado na criatura de número 3. Por conta disso, essa propriedade terá o `?` ao lado do tipo, representando que o Kotlin deve permitir que essa variável aceite valores nulos.

Antes de criar a classe, navegue até o pacote `model` e crie um pacote chamado `source.remote.entity`. Dentro do pacote, crie a classe `CreatureApiResponse` e adicione a seguinte estrutura:

```
package
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.source.remote.en
tity

data class CreatureApiResponse(
    val number: Int,
    val name: String,
    val image: String,
```

```
    val legendary: Boolean,
    val evolveTo: CreatureApiResponse?,
)
```

Preparando o Service

Com a entidade criada, vamos realizar o mapeamento do *endpoint* para compor o `Service`. Geralmente, temos um único `Service` para agrupar *endpoints* similares. No entanto, nesse caso temos apenas um *endpoint*, que lista todas as criaturas. Para iniciar a criação do `CreatureService`, clique com o botão direito em cima do pacote `source.remote.entity` e crie um pacote `remote.services`. Note que, para criar esse pacote, o nome `entity` deve ser apagado, mantendo `services` no lugar.

Dentro do pacote criado, crie uma interface chamada `CreatureService`, como no código a seguir:

```
package
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.source.remote.se
rvices

interface CreatureService {
}
```

Como o *endpoint* `creature` retorna a lista de criaturas, podemos chamá-lo de `findAll`. Vamos criar um método com o nome que preferirmos e colocar a estrutura de dados que foi mapeada como tipo de retorno do método, envolvida pelo tipo `Call`, que sinaliza que esse método retorna uma chamada do Retrofit:

```
// ...
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.source.remote.en
tity.CreatureApiResponse
import retrofit2.Call

interface CreatureService {
```

```
  fun findAll(): Call<List<CreatureApiResponse>>
}
```

O próximo passo para mapear um *endpoint* é inserir o método HTTP utilizado. Ao abrir o navegador na URL do *endpoint*, estamos realizando uma requisição HTTP do tipo `GET`. Portanto, vamos adicionar a *annotation* `@GET("creature")` informando o caminho do *endpoint*:

```
// ...
import retrofit2.http.GET

interface CreatureService {
    @GET("creature")
    fun findAll(): Call<List<CreatureApiResponse>>
}
```

Agora que temos o *endpoint* mapeado, vale lembrar que estamos utilizando o RX na aplicação, que visa facilitar a utilização de chamadas assíncronas. Quando esse é o caso, o Retrofit possui integração nativa com o RX, basta alterar o tipo `Call` pelo `Observable`. Dessa forma, o Retrofit prepara um `Observable` que, ao ser observado por um `Subscriber`, imediatamente dispara a requisição:

```
// ...
import io.reactivex.rxjava3.core.Observable

interface CreatureService {
    @GET("creature")
    fun findAll(): Observable<List<CreatureApiResponse>>
}
```

Pronto! Com isso, temos o tipo de retorno da API e o `Service` devidamente criados e prontos para serem utilizados. O próximo passo é a criação do Remote Data Source do *endpoint* `Creature`.

Criando o Remote Data Source

Com a configuração do *endpoint* pronta, precisamos de um local que disponibilize o acesso às informações pelas outras classes do sistema. Uma estratégia muito interessante nesses casos é evitar trabalhar com as estruturas de dados da API, pois além de essas estruturas serem pouco flexíveis, uma vez que dependem de mudanças na API, elas também são muito frágeis, pois alterações na API podem quebrar comportamentos da aplicação difíceis de corrigir em uma estrutura pouco flexível.

Para contornar esse problema, vamos utilizar a entidade de domínio `Creature`, que já é amplamente utilizada na aplicação. Portanto, o `CreatureService` deve expor uma lista de criaturas que obtém os dados através da API, mas converte a estrutura de dados da API `CreatureApiResponse` para a estrutura de domínio `Creature`.

A única forma de conseguir realizar requisições na API é através do `Service`, que até então é uma interface que não foi implementada. Como o Retrofit cuida da implementação das interfaces, reduzindo a quantidade de *boilerplate* necessário, precisamos utilizar a declaração `Retrofit.Builder()` para construir a instância do Retrofit e utilizá-la na implementação dos `Services`.

Começaremos criando a classe `CreatureRemoteDataSource` dentro do pacote `source.remote`, marcando-a como `@Singleton` e disponibilizando-a para injeção de dependência com a declaração `@Inject` no `constructor()`, da seguinte forma:

```
package
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.source.remote

import javax.inject.Inject
import javax.inject.Singleton

@Singleton
class CreatureRemoteDataSource @Inject constructor() {
}
```

Dentro da classe, vamos preparar a instância do Retrofit logo no momento em que a classe é inicializada, através do bloco de inicialização `init { }`. Essa instância pode ser obtida através da declaração `Retrofit.Builder()`, informando a base da URL e o `ConverterFactory` da biblioteca de desserialização (no caso, o GSON). Como estamos usando o `Observable` como tipo de retorno da API, substituindo o `Call` do Retrofit, também é necessário adicionar o `CallAdapterFactory` do RX. Com tudo devidamente declarado, basta executar o método `build()` para obtenção da instância, como no trecho a seguir:

```
// ...
import retrofit2.Retrofit
import retrofit2.adapter.rxjava3.RxJava3CallAdapterFactory
import retrofit2.converter.gson.GsonConverterFactory

@Singleton
class CreatureRemoteDataSource @Inject constructor() {
    init {
        val retrofit = Retrofit.Builder()
            .baseUrl("https://devmon-api.onrender.com/")
            .addCallAdapterFactory(RxJava3CallAdapterFactory.create())
            .addConverterFactory(GsonConverterFactory.create())
            .build()
    }
}
```

Com a instância do Retrofit pronta, podemos utilizá-la para criar uma instância do `Service`. Vale ressaltar que, para que a instância do `Service` seja utilizada por propriedades e métodos, precisamos que ela seja uma propriedade da classe. Colocamos a propriedade como `private` para evitar que outras classes utilizem o `Service` diretamente; assim, evitamos que operações de API sejam disparadas diretamente por outras partes do código, o que dificultaria a manutenção e poderia provocar chamadas de network desnecessárias.

Para criar a instância do `Service`, utilizaremos o método `retrofit.create()`:

```
class CreatureRemoteDataSource @Inject constructor() {
    private val service: CreatureService

    init {
        val retrofit = Retrofit.Builder()
            .baseUrl("https://devmon-api.onrender.com/")
            .addCallAdapterFactory(RxJava3CallAdapterFactory.create())
            .addConverterFactory(GsonConverterFactory.create())
            .build()

        service = retrofit.create(CreatureService::class.java)
    }
}
```

Agora que o `service` está instanciado, vamos expor o `Observable` contendo uma `List<Creature>`, ou seja, a lista de criaturas obtidas através da API, mas disponibilizadas no formato da camada de domínio:

```
// ...
class CreatureRemoteDataSource @Inject constructor() {
  // ... (service & init)

  val creatures: Observable<List<Creature>> =
    service.findAll().map { list ->
      list.map { it.toDomain() }
    }

  private fun CreatureApiResponse.toDomain(): Creature {
    return Creature(
      number = number,
      name = name,
      imageUrl = image,
    )
  }
}
```

Integrando a API com o restante da aplicação

Agora que as classes que disponibilizam as informações da API estão criadas, vamos mudar a maneira com que o `CreaturesRepository` prepara o `Observable` com a lista de criaturas.

Primeiro, precisamos acessar o `RemoteDataSource` na classe `CreaturesRepository`. Para isso, basta declarar a propriedade direto no `constructor` para que o Hilt faça a injeção da dependência, da seguinte forma:

```
// ...
import
tech.salvatore.livro_android_kotlin_paulo_salvatore.model.source.remote.Cr
eatureRemoteDataSource

class CreaturesRepository @Inject constructor(
    remoteDataSource: CreatureRemoteDataSource
) {
  // ...
```

Com a instância do `remoteDataSource` disponibilizada, podemos acessar a propriedade `creatures`, recebendo o `Observable<List<Creature>>`.

Para disponibilizar a lista de criaturas, um caminho possível seria fazer com que a propriedade `CreaturesRepository.creature` receba diretamente o valor de `remoteDataSource.creatures`, pois ambas possuem o tipo `Observable<List<Creature>>`. No entanto, o principal problema com essa abordagem é que toda vez que a propriedade `creatures` fosse utilizada por alguma implementação, uma nova requisição seria disparada, fazendo com que várias requisições fossem feitas desnecessariamente.

Uma ótima forma de contornar esse problema quando estamos usando o RX é utilizando o `ReplaySubject` em vez do `Observable`. O que o `ReplaySubject` faz é armazenar os últimos eventos emitidos por um `Observable`, dependendo de uma quantidade configurável. Assim,

podemos criar um `ReplaySubject` que repete apenas o último evento, fazendo com que a lista de criaturas sempre esteja disponível para a aplicação, sem precisar de novas requisições.

> Vale ressaltar que o `ReplaySubject`, por baixo dos panos, é um `Observable`. Além do mecanismo de replay, o `ReplaySubject` entra na categoria de `Subjects`, que são `Observables` comuns, com a exceção de que o método `onNext` pode ser chamado depois de sua criação. Isso faz com que `Subjects` sejam muito úteis quando estamos criando `Observables`, mas queremos deixar a lógica de atualização dos eventos separada.

Para criar um `ReplaySubject`, utilizamos a declaração `ReplaySubject.create(1)`, onde `1` é a quantidade de eventos que queremos guardar:

```
// ...
import io.reactivex.rxjava3.subjects.ReplaySubject

class CreaturesRepository @Inject constructor(
  remoteDataSource: CreatureRemoteDataSource
) {
  val creatures: ReplaySubject<List<Creature>> = ReplaySubject.create(1)
  // ...
}
```

O valor da propriedade `creatures` pode ser inicializado no `init { }`, criando um `subscribe` no `remoteDataSource.creatures` e repassando o valor via `onNext()`. Dessa forma, a requisição só é feita durante a inicialização do `CreaturesRepository`. Vamos apagar todo o conteúdo do bloco `init { }` e substituir pelo conteúdo a seguir, adicionando alguns `Logs` para informar se a requisição foi bem-sucedida ou não.

```
class CreaturesRepository @Inject constructor(
  remoteDataSource: CreatureRemoteDataSource
) {
  val creatures: ReplaySubject<List<Creature>> = ReplaySubject.create(1)
```

```
  init {
    remoteDataSource.creatures
      .doOnNext {
        Log.d("CREATURES_REPOSITORY", "${it.size} creatures were loaded
from API.")
      }
      .doOnError {
        Log.e("CREATURES_REPOSITORY", "Error loading creatures from API.",
it)
      }
      .subscribe {
        creatures.onNext(it)
      }
  }
  // ...
}
```

Com a propriedade `CreaturesRepository.creatures` preparada, podemos executar o código para testar como a requisição é feita. Antes de executar o app, abra a URL do back-end em algum navegador, para garantir que o servidor está online e funcionando, como próxima imagem.

Figura 11.6: Servidor rodando.

Para visualizar a ordem do processamento da requisição, vou colocar alguns *breakpoints* na chamada do `remoteDataSource.creature`, de acordo com a imagem a seguir.

> O breakpoint é um ponto de parada na execução do código. Antes de a aplicação executar aquela linha de código, o breakpoint vai parar a execução e nos permitir visualizar os valores de variáveis naquele momento, executar algumas declarações simples e algumas outras ferramentas interessantes para depuração do código.
>
> Para adicionar um *breakpoint*, basta clicar ao lado direito do número da linha. Para remover, basta clicar na bolinha vermelha.

Figura 11.7: Breakpoints no RemoteDataSource.

Para que os *breakpoints* sejam ativados, execute a aplicação clicando no botão `Debug app`. Assim que o app inicia, o primeiro *breakpoint* é acionado. Como a aplicação está inicializando as classes, o app ainda não está pronto e exibe a tela de carregamento. Note que o bloco `init { }` está sendo executado na *Main Thread*, que é onde todo o processamento da UI do Android ocorre. Nesse momento, estamos apenas declarando o que o `Observer` do `Observable` `creatures` fará quando algum evento for emitido.

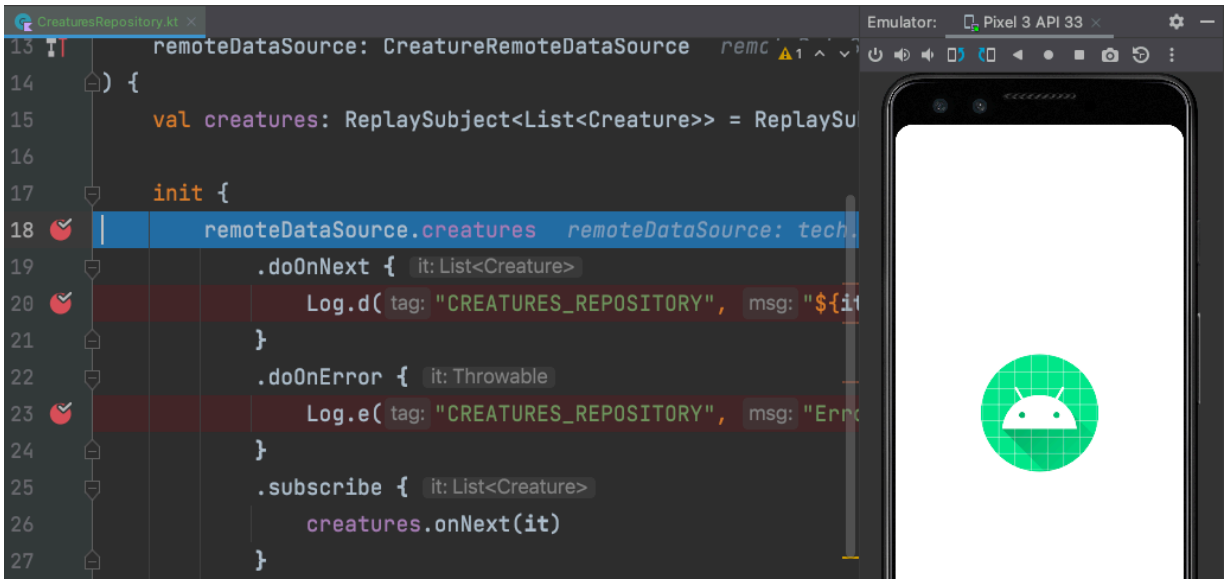


Figura 11.8: Primeiro breakpoint acionado.

Na janela de `Debug`, é possível visualizar que estamos na *thread* chamada `main` (primeiro nome que aparece), como mencionado anteriormente. Para avançar para a próxima linha, podemos clicar no botão `Resume program`, de acordo com a imagem a seguir.

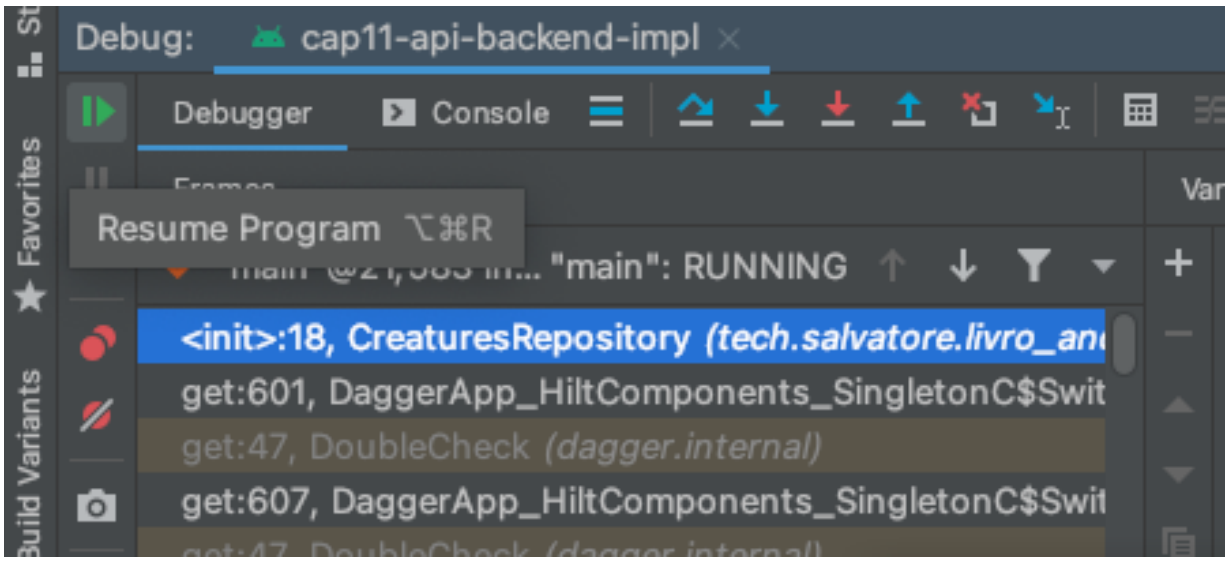


Figura 11.9: Breakpoint - Resume Program.

Ao retomar o programa, o `Observer` já está declarado e aguardando a emissão de eventos. Após algum tempo, assim que a requisição completa, o próximo *breakpoint* é executado.

OBSERVAÇÃO: caso o *breakpoint* de erro seja disparado nesse passo, verifique se o servidor de back-end está online e se os códigos que você escreveu estão corretos. Também é possível resumir o programa para que o log de erro seja devidamente registrado. A partir disso, olhe o Logcat com a tag `CREATURES_REPOSITORY` para buscar mais informações sobre o que ocorreu.

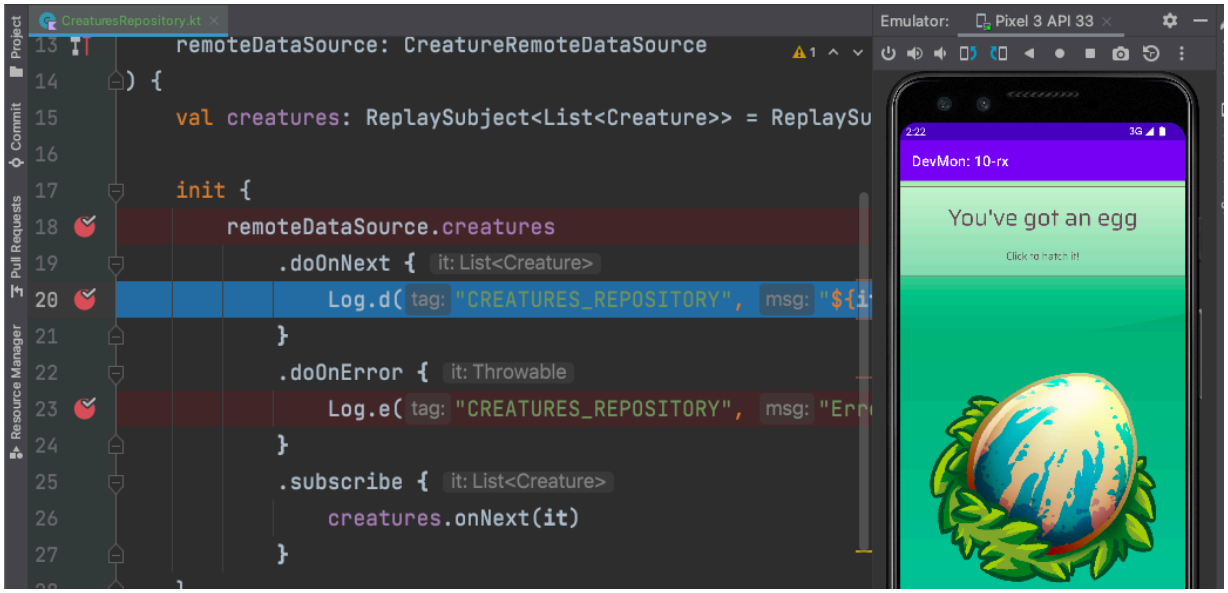


Figura 11.10: Segundo breakpoint rodando.

Nesse momento, a primeira tela da aplicação já foi carregada e estamos em uma nova *thread* chamada `OkHttp`, que é a biblioteca que o Retrofit utiliza para realizar as operações na internet. Note que é possível visualizar a lista de criaturas na propriedade `it`.

Ao retomar o programa, a lista de criaturas é enviada para o `ReplaySubject`, que manterá os valores armazenados e disponíveis sempre que algum local do sistema precisar. Com isso, além de manter o valor da propriedade `creatures` sempre disponível, garantimos que a requisição para a API seja realizada no momento que a classe `CreaturesRepository` é inicializada.

Para depurar as operações de network, podemos utilizar o `Network Inspector`, localizado na aba `App Inspection`, como mostra a imagem a seguir. No entanto, o Network Inspector leva um tempo para começar a exibir as requisições, então as primeiras requisições não são mostradas.

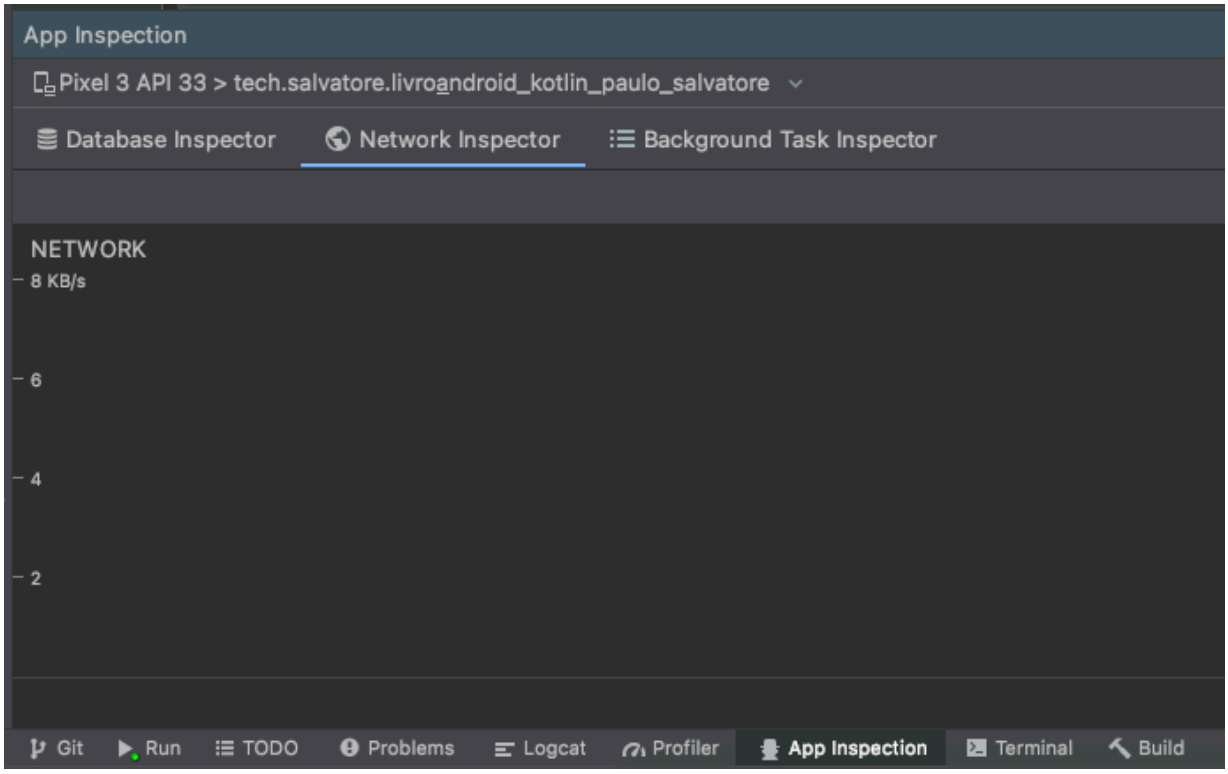


Figura 11.11: Network Inspector.

Caso queira visualizar a primeira requisição para a lista de criaturas, é possível adicionar um `Timer`, da seguinte forma:

```
// ...
import java.util.*
import kotlin.concurrent.schedule

@Singleton
class CreaturesRepository @Inject constructor(
  // ...
  init {
    Timer().schedule(5000) {
      remoteDataSource.creatures
          // ...
    }
  }
  // ...
}
```

Rode a aplicação e note que a requisição passa a ser exibida. É possível selecioná-la para exibir mais detalhes, como na imagem:

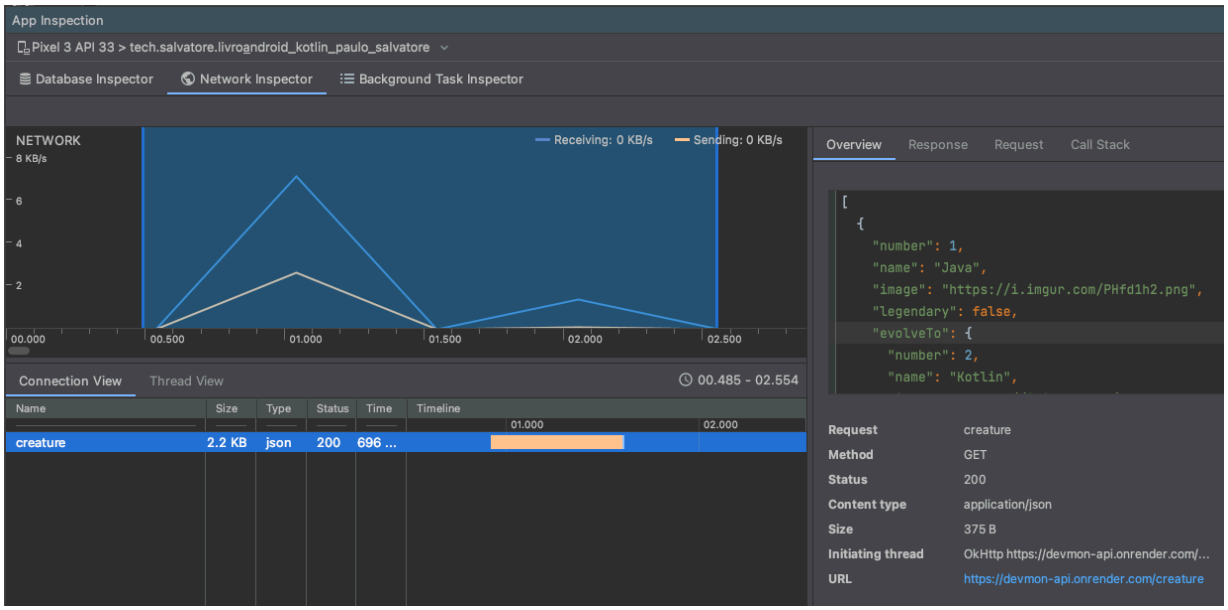


Figura 11.12: Informações da requisição.

Após os testes, remova a declaração do `Timer().schedule()` e certifique-se de remover os `imports` sem utilização também.

Pronto! Com isso, temos a implementação do back-end no nosso app. Como a adaptação para RX já havia sido feita, implementar o back-end foi simples como trocar um plug por outro.

Conclusão

Neste capítulo, foi possível entender como funciona o Networking no Android, além de entendermos um pouco mais sobre o uso da biblioteca Retrofit para realização das requisições. Também vimos que o JSON é o principal formato de estrutura de dados. Ele utiliza a biblioteca GSON para realizar o processo de desserialização da informação, transformando-a em objetos na memória do dispositivo. Por fim, fizemos a implementação do back-end em nosso projeto, garantindo que a lista de criaturas será carregada a partir de um serviço na internet.

Por último, mas não menos importante, chegou a hora de revelar que o nome do DevMon #11 é o **SQL** e o DevMon #12 é o **NoSQL**!

SQL é o nome de uma famosa tecnologia de bancos de dados relacionais, utilizada há muitas décadas e aplicada em diversos dialetos, como MySQL, PostgreSQL, SQLite, MariaDB, OracleDB, entre outros. Quando precisamos desenvolver uma aplicação que possui uma estrutura de dados muito bem mapeada e os relacionamentos entre as entidades bem claros, os bancos relacionais com SQL são uma ferramenta muito poderosa.

Já **NoSQL** é o nome de uma tecnologia de banco de dados que significa *not-only SQL*, ou seja, engloba outros modelos de bancos de dados além do modelo relacional. Por possuir muitas implementações diferentes, os bancos de dados NoSQL acabam tendo características muito diferentes entre si, como os modelos orientados a documento (ex.: MongoDB), modelos orientados a chave-valor (ex.: Redis), entre outras soluções como CassandraDB, Neo4j, Google BigQuery etc.

URL das imagens https://i.imgur.com/4d8PVxQ.png e https://i.imgur.com/YEuaYEZ.png

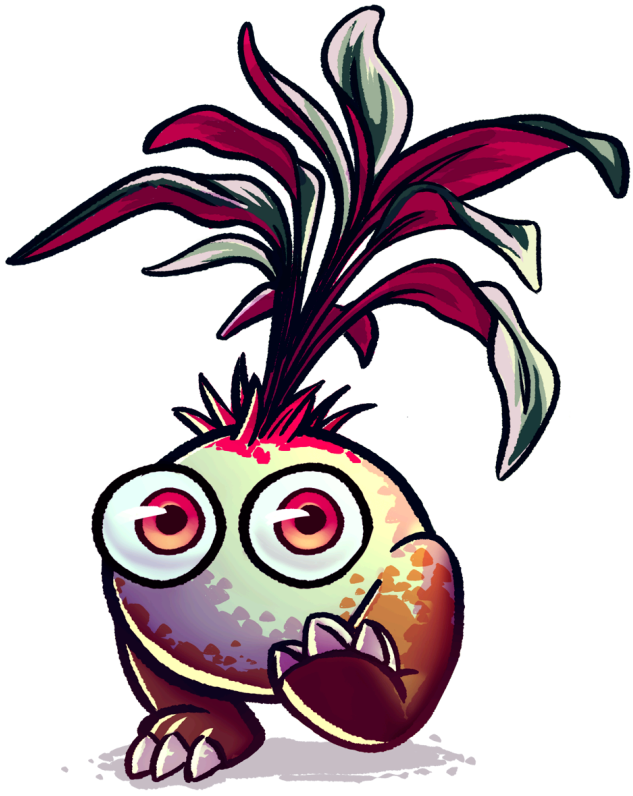


Figura 11.13: O DevMon #11 se chama SQL!

Figura 11.14: O DevMon #12 se chama NoSQL!

Capítulo 12
Encerramento

Chegamos ao final do livro! Foi uma jornada e tanto, não?! Vamos falar um pouco sobre o que foi aprendido durante o livro e apresentar quais os próximos passos que você pode dar a partir do que aprendemos.

A última criatura da nossa coleção é uma criatura lendária, o DevMon #13! O nome da criatura é em homenagem a um motor de jogos muito famoso e que utiliza a linguagem de programação C# como base para construção de scripts. Jogos famosos como Overcooked!, Assassin's Creed, Pokémon Go™, entre outros, foram desenvolvidos nessa game engine, que é gratuita para quem quiser começar a desenvolver. E aí, já sabe qual é?

Figura 12.1: Qual é o DevMon #13?

O livro abordou diversos tópicos importantes para o desenvolvimento de aplicativos Android nativos, incluindo o uso da linguagem Kotlin e do padrão MVVM, bem como a utilização de bibliotecas como Navigation, Retrofit, GSON, Hilt, RX e Glide. Os capítulos do livro cobriram tópicos como criação do projeto, fluxo de navegação, lista de criaturas, *Data Binding*, aquisição e visualização de criaturas e integração com back-end.

Ao final do livro, os leitores e as leitoras devem ter um bom entendimento dos conceitos avançados de desenvolvimento Android, bem como uma ampla variedade de ferramentas e técnicas para aplicar em seus projetos futuros. Além disso, o projeto final do livro deve servir como um exemplo prático de como esses conceitos podem ser aplicados em um projeto real.

Espero que tenha curtido o projeto que desenvolvemos até aqui e que tenha aprendido e relembrado diversos conceitos. Apesar de as implementações do livro se encerrarem por aqui, você pode conferir outras funcionalidades no módulo `project-complete-with-extras` e, também, nos vídeos do meu canal no YouTube (https://youtube.com/PauloSalvatore).

Figura 12.2: Canal do YouTube do Paulo Salvatore.

Ter criado e publicado esse projeto foi uma experiência gratificante pra mim. Espero que este livro tenha sido útil em seu caminho para se tornar uma pessoa desenvolvedora Android experiente e bem-sucedida. Compartilhe com as pessoas ao seu redor o que você achou deste livro e os conhecimentos aqui adquiridos.

Por último, mas não menos importante, chegou a hora de revelar que o nome do DevMon #13 é **Unity**! Um dos motores de jogos (*engine*) mais populares do cenário, Unity é completamente gratuito para quem quiser desenvolver games e é uma ótima oportunidade para aprender a linguagem de programação C#, que é muito parecida com Java/Kotlin em alguns aspectos, mas com semelhanças

também ao JavaScript/TypeScript. Por ser uma aplicação multiplataforma, é possível desenvolver soluções que rodam em diversas plataformas como Android, iOS, Windows, Linux, Web, PlayStation, Switch, entre outras.

URL da imagem: https://i.imgur.com/jyo3rpf.png

Figura 12.3: O DevMon #13 se chama Unity!

12.1 Próximos passos?

Embora o livro tenha chegado ao fim, o aprendizado e o desenvolvimento de habilidades em Android nunca terminam. Existem sempre novas técnicas e tecnologias sendo lançadas; por isso, é importante continuar a aprender e se atualizar.

Caso queira continuar a desenvolver mais a aplicação e aplicar novos conhecimentos, sugiro as seguintes implementações:

Novas funcionalidades

- Evolução dos DevMon;
- Mecanismo de cuidar dos DevMon, adicionando as ações "brincar", "alimentar" e "treinar".

Tecnologias

- **Android Room**, para utilização de banco de dados SQLite no próprio dispositivo;
- **DataStore**, para armazenar e persistir dados do usuário atual da aplicação.